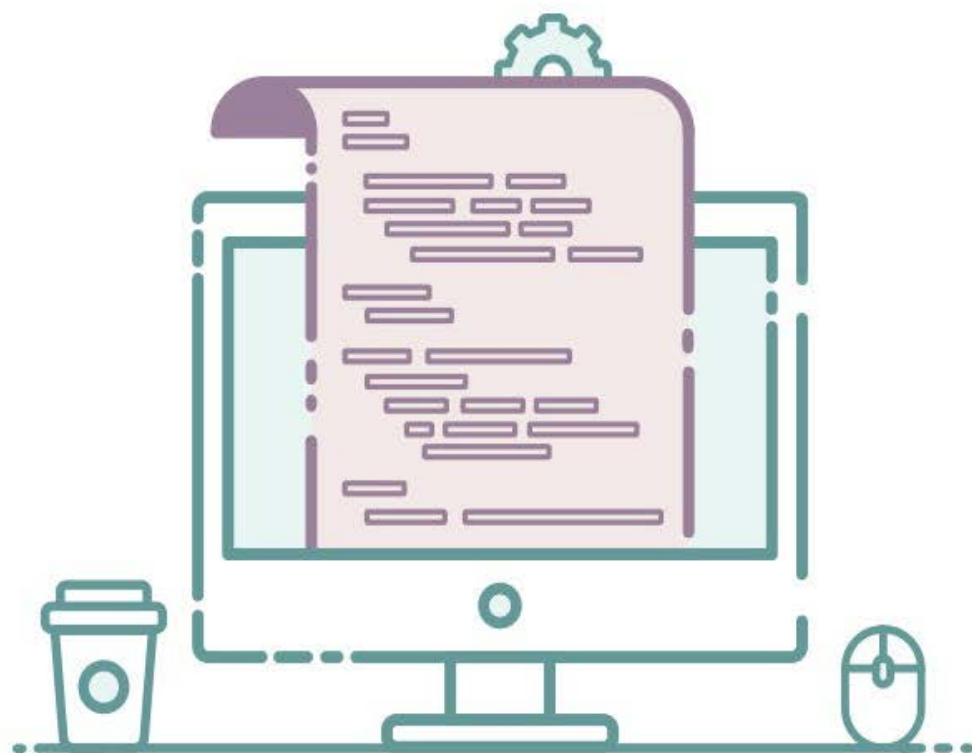


Turbine seu CSS

Folhas de estilo inteligentes
com Sass





ISBN

Impresso e PDF: 978-85-5519-225-8

EPUB: 978-85-5519-226-5

MOBI: 978-85-5519-227-2

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

PREFÁCIO

Este livro foi escrito com o intuito de desmistificar o uso de pré-processadores para a escrita de folhas de estilo, especificamente o Sass. Nele você vai encontrar os motivos para optar pelo uso de um pré-processador e explorar suas principais funcionalidades: variáveis, mixins, aninhamento, herança, funções e modularização.

Qualquer pessoa está convidada para ler o livro. No entanto, para um melhor aproveitamento, é importante que você tenha familiaridade na construção de páginas web e que se sinta confortável com o uso de CSS.

Este é um livro que pode ser considerado técnico e prático. No transcorrer dos capítulos, você será apresentado às funcionalidades, e imergirá em exemplos reais e úteis de cada uma delas. Além disso, uma aplicação web será desenvolvida e incrementada a cada capítulo com os itens apresentados, mostrando a necessidade de cada um deles.

COMO O LIVRO ESTÁ ORGANIZADO

No primeiro capítulo, você verá os pré-processadores, compreendendo as vantagens e desvantagens do seu uso. Neste capítulo também será configurado o ambiente de trabalho e definições básicas, como extensão de arquivos, compilação e configuração de saída.

O capítulo 2 discorre sobre as variáveis. Nele, você vai aprender a definir e usar uma variável, além de entender quais momentos são oportunos para o seu uso. Já o capítulo 3 fala sobre o aninhamento. Neste, você também terá várias formas de aproveitar este recurso e compreender que o aninhamento em excesso pode ser muito

prejudicial às suas folhas de estilo.

Nos capítulos 4 e 5, você verá como reutilizar blocos de código com o uso de mixins e herança. Nestes capítulos, além de ser apresentado a essas duas funcionalidades do Sass, padrões de código como o DRY e o OOCSS serão explorados com o intuito de facilitar o reúso e a manutenção do seu código.

O capítulo 6 mostra como modularizar o seu stylesheet usando o Sass. Particionamento e importação de arquivos são assuntos deste capítulo. Uma técnica chamada SMACSS também é abordada.

O capítulo 7 é uma festa das cores. Nele você terá acesso a inúmeras funções de cores disponíveis pelo Sass, além de montar temas de cores dinamicamente.

Diria que os capítulos 8 e 9 complementam-se. No capítulo 8 serão apresentados os tipos de dados e operações disponíveis que, na grande maioria das vezes, são usados em conjunto com as estruturas de controle e repetição — `if/else`, `for`, `while`, `each` — levantadas no capítulo 9.

No capítulo 10, você conhecerá um pouco do ecossistema Sass e também terá dicas de como aprofundar os seus estudos.

Acesse o código do livro e discuta com a gente

Os exemplos e o projeto apresentado neste livro estão disponíveis para download em:

- <https://github.com/rosicleiafrasson/livro-sass-exemplos>
- <https://github.com/rosicleiafrasson/livro-sass-projeto>

É claro que você deve usá-los apenas como fonte de consulta, pois a digitação do código aumenta significativamente o aprendizado.

SOBRE O AUTOR

Meu nome é Rosicléia Frasson, sou formada em Ciência da Computação e trabalho com desenvolvimento de software a alguns anos. Durante essa caminhada, já trabalhei com requisitos, análise e desenvolvimento.

Já atuei como professora e ministrei aulas de programação, engenharia de software, processos de desenvolvimento e qualidade de software. Também já ministrei cursos in-company. Escrever e ensinar sempre foram uma grande paixão.

Já trabalhei também em algumas empresas de desenvolvimento e, atualmente, sou desenvolvedora na Thomson Reuters. Posso dizer que nesses anos de trabalho, meu foco maior sempre foram as linguagens de programação: Java, C, PHP e JavaScript. No entanto, fiz parte de um projeto, no qual escrevi um framework de componentes. Nele fui apresentada ao Sass e mergulhei de cabeça no front-end.

Este livro retrata um pouco do que aprendi nesta jornada. Espero que o conhecimento adquirido com ele seja tão útil para você como foi para mim.

Para finalizar, deixo meu contato para sugestões, críticas ou para bater um papo mesmo: rosicleiafrasson@gmail.com

Sumário

1 Primeiros passos com Sass	1
1.1 Por que usar um pré-processador?	1
1.2 Por que o Sass?	2
1.3 Configurando o ambiente de trabalho	5
1.4 Processo de compilação	7
1.5 Configuração de saída	11
1.6 Usando comentários	13
1.7 O que esperar dos próximos capítulos	14
2 Reúse valores de propriedades com variáveis	15
2.1 Declarando uma variável	16
2.2 Referencie uma variável	19
2.3 Variáveis globais e de escopo	21
2.4 Como nomear variáveis	22
2.5 Faça muito mais com variáveis	22
2.6 Um projeto real	24
2.7 O que esperar do próximo capítulo	29
3 Escreva seletores longos mais rápido com aninhamento	31
3.1 Como aninhar regras de estilo	31
3.2 Usando o & para concatenar	34

3.3 Aninhamento de propriedades	38
3.4 Aninhamento de grupos de seletores	41
3.5 Combinação de filhos e irmãos (>, + e ~)	41
3.6 Perigos do aninhamento excessivo	46
3.7 Um projeto real	49
3.8 O que esperar do próximo capítulo	52
4 Reúse blocos de estilo com mixins	53
4.1 Definindo um mixin	53
4.2 Incluir um mixin	55
4.3 Uso de parâmetros	58
4.4 Mixin vendor prefix	62
4.5 Regras CSS nos mixins	63
4.6 Fazendo muito mais com mixins	64
4.7 Um projeto real	66
4.8 O que esperar do próximo capítulo	69
5 Evite propriedades duplicadas com herança	70
5.1 Usando a herança	70
5.2 Placeholder	73
5.3 A relação íntima entre herança e OOCSS	76
5.4 Herança versus mixin	81
5.5 Um projeto real	82
5.6 O que esperar do próximo capítulo	85
6 Dividir para conquistar	86
6.1 Criando um arquivo particionado	87
6.2 Importando uma parte de um arquivo	88
6.3 Excelentes candidatos para arquivo particionado	90
6.4 Usando particionamento para arquitetar seu CSS	93
6.5 Um projeto real	101

6.6 O que esperar do próximo capítulo	103
7 Use o poder das cores	105
7.1 Sistemas de cores	106
7.2 A função rgba	108
7.3 Retornando a composição de uma cor	109
7.4 Mix de cores	110
7.5 Escurecendo e clareando cores	111
7.6 Criando cores inversas e complementares	113
7.7 Saturar, dessaturar	115
7.8 Ajustar o matiz de uma cor	116
7.9 Adicionando canal alpha	117
7.10 Em tons de cinza	118
7.11 Um projeto real	119
7.12 O que esperar do próximo capítulo	123
8 Desvendando e manipulando os tipos de dados	124
8.1 Numbers	125
8.2 Strings	126
8.3 Booleans	126
8.4 Lists	127
8.5 Maps	128
8.6 Operações	130
8.7 Funções	134
8.8 Um projeto real	136
8.9 O que esperar do próximo capítulo	142
9 Obtendo controle extremo sobre o código gerado	144
9.1 Escrevendo estruturas de decisão com @if e @else e @else if	
9.2 Montando uma estrutura de controle com @for	151 ¹⁴⁴
9.3 Controlando a saída de código com @while	158

9.4 Montando classes dinamicamente com @each	160
9.5 Um projeto real	162
9.6 O que esperar do próximo capítulo	166
10 Indo além	168
10.1 Ecossistema Sass	168
10.2 Não pare por aqui	170
10.3 Palavras finais	171

PRIMEIROS PASSOS COM SASS

1.1 POR QUE USAR UM PRÉ-PROCESSADOR?

Até pouco tempo atrás, estilizar uma página web era uma tarefa relativamente simples. No entanto, a web evoluiu. Tudo mudou. As páginas agora são feitas pensando na melhor experiência do usuário.

Hoje, os meios de acesso são diversos, incluindo smartphones, tablets, smartTVs. Os sites que eram simples foram ficando complexos. A web que até outrem era ocupada em sua grande maioria por sites institucionais, agora virou plataforma para grandes aplicações corporativas. Enfim, o aumento da diversidade de uso e de acesso na web tornou a tarefa de escrever CSS muito mais complexa.

O CSS é uma linguagem extremamente simples e muito poderosa. No entanto, é uma linguagem declarativa, não uma linguagem de programação. Sendo assim, não temos os recursos que uma linguagem de programação fornece como declaração de variáveis para facilitar a manutenção, uso de métodos/funções que evitem duplicação de código, pouco ou quase nenhum suporte para operações matemáticas, possibilidade de escrever código aninhado e um meio para importação de arquivos.

No cenário em que vivemos, onde o mundo está cada vez mais

competitivo e é necessário garantir entregas rápidas para continuar no mercado, precisamos deixar as folhas de estilo mais flexíveis, mais fáceis de manter, escaláveis e mais rápidas para serem produzidas. Por estes motivos, é que surgiram os pré-processadores.

Os pré-processadores funcionam como uma espécie de extensão do CSS nativo e auxiliam a criação de folhas de estilo com a adição de funcionalidades. Eles ajudam a diminuir a repetição de código e permitem implementar mudanças de forma mais rápida. O grande benefício do seu uso é que as folhas de estilo são escritas mais rapidamente e podem ser mantidas com menos esforço.

Existe uma série de pré-processadores no mercado. Dentre os mais conhecidos estão o Sass, o Less e o Stylus. Estes compartilham diversos recursos e possuem suas particularidades. Neste livro, estaremos abordando os conceitos básicos dos pré-processadores, e vamos nos aprofundar na sintaxe e funcionalidades disponibilizadas pelo Sass.

O Sass — Syntactically Awesome StyleSheets — segundo o fabricante é uma extensão do CSS que adiciona potência e elegância ao CSS nativo. O Sass possui recursos como variáveis, aninhamento de regras, mixins e suporte a operações matemáticas com sintaxe compatível ao CSS. Ele ajuda a manter as folhas de estilo organizadas e facilita a manutenção.

1.2 POR QUE O SASS?

Essa é uma resposta da qual o gosto pessoal é o que prevalece. Existem inúmeros artigos e discussões acaloradas em fóruns, comparando diversos pré-processadores e elencando vantagens e desvantagens de cada um. Eu, particularmente quando precisei usar pré-processamento para folhas de estilos pela primeira vez, fiz um comparativo com pontos que considerava importantes e achei que o

Sass me atenderia melhor. E me atendeu perfeitamente.

Para justificar a minha escolha, a seguir estão os critérios que foram considerados antes de optar pelo Sass.

- Possibilidade de modularização das folhas de estilo
- Curva de aprendizado
- Documentação
- Qualidade de código gerado
- Envolvimento da comunidade
- Capacidade de reaproveitamento de código

Bom, o ponto de partida foi a aderência do mercado. Grandes companhias incluindo o eBay, BBC, Instagram, LinkedIn, Square e Groupon usam Sass para produzir seu CSS. Mas o principal motivo que me fez pensar em deixar de escrever CSS nativo foi a impossibilidade de reaproveitar código.

Incomodava-me bastante o fato de precisar usar uma mesma cor em vários pontos distintos da folha de estilo e ter de definir essa cor em todos os pontos. Uma simples mudança de cor proposta pela equipe de design significava uma grande procura em um emaranhado de código.

Isso sem contar com a grande quantidade de código repetido muitas vezes com pouquíssimas alterações, que dificultava muito a manutenção. Diante disso, eu precisava de um mecanismo que possibilitasse o uso de variáveis, definição de procedimentos, uso de herança, funções matemáticas e que implementasse os conceitos bases da lógica de programação. O Sass foi quem me atendeu melhor esse conjunto de requisitos.

Um outro aspecto importante a ser considerado era a possibilidade de particionamento de arquivos. Embora o CSS nativo tenha uma forma bem rudimentar de fazer o particionamento —

que será abordada nos tópicos seguintes —, eu precisava de algo que garantisse a modularização sem acabar com a performance. E mais, eu precisava que a modularização fosse facilmente compreendida, já que muitas pessoas mantêm esse código constantemente.

No Sass, encontrei o que eu precisava para o particionamento e importação de arquivos. Quando decidi utilizar um pré-processador, o projeto em que trabalhava já possuía uma enorme quantidade de regras de estilo escritas e não existia tempo hábil para reescrever todas.

Dessa forma, era necessário suportar o legado. Uma das sintaxes suportadas pelo Sass é baseada na sintaxe padrão do CSS. Isso garante que todo o código escrito em CSS nativo é suportado pelo Sass.

Como já mencionei, muitas pessoas eram responsáveis por alterações nos estilos das páginas. Muitas delas mal conheciam o CSS nativo, e era necessário que a mudança não impactasse na produtividade. Precisávamos de um mecanismo com grande facilidade de aprendizado e com documentação abundante.

O Sass, além de possuir sua documentação oficial, possui uma vasta quantidade de artigos espalhados pela rede. Isso facilita bastante o seu uso.

Sempre que vamos utilizar uma ferramenta, um framework ou uma tecnologia diferente das que usamos habitualmente, é natural sentir receio quanto a sua estabilização e melhorias futuras. O envolvimento da comunidade é crucial para o aprimoramento de uma tecnologia. Por este motivo, é importante levantar números relacionados a quantidade de issues, pull requests e commits. Os números colhidos em relação ao Sass foram bem satisfatórios.

Por fim, os browsers só entendem CSS. Embora uma meta

linguagem esteja sendo usada para escrever as regras de estilo, no fim de tudo essas regras precisam ser traduzidas para que o navegador possa processá-las, e a qualidade desse código gerado é muito importante. O compilador do Sass possui uma série de configurações para a personalização desse código de saída e o agrupamento de um conjunto delas me agradou bastante.

UM POUCO DE HISTÓRIA

O Sass começou a ser criado em 2006 por Hampton Catlin e tem como seus principais contribuintes Nathan Weizembaum e Christopher Eppstein. O processador do Sass é escrito em Ruby, no entanto, não é necessário ter conhecimento na linguagem para poder utilizá-lo.

Sass é open source e seu diretório de desenvolvimento está hospedado em: <https://github.com/sass>. Podemos contribuir para a sua melhoria com relato de bugs, solicitações de recursos e codificação.

Atualmente, o Sass conta com uma série de projetos extensivos que aumentam significativamente suas funcionalidades, como Compass, Susy e Bourbon. Sua grande aceitação por parte dos desenvolvedores colaborou para o surgimento de muitos outros pré-processadores. Os mais conhecidos são o Less e o Stylus. Se você quiser pesquisar mais a respeito do Less e do Stylus antes de tirar suas conclusões, fica a sugestão de alguns links: <http://lesscss.org/>, <http://stylus-lang.com/> e <https://css-tricks.com/sass-vs-less/>.

1.3 CONFIGURANDO O AMBIENTE DE

TRABALHO

Quando as regras de estilo são escritas usando o Sass, é necessário transformá-las em CSS para que o navegador consiga renderizar. Quando optamos pelo Sass, existem muitas maneiras de fazer essa transformação, e a opção por uma delas depende do seu nível de conhecimento e do tipo de projeto que você está tentando fazer.

Se você pretende aprender ou fazer alguns testes pode optar pelo editor online *CodePen*. Com ele, é possível criar, compartilhar e pesquisar exemplos de código.

O editor é dividido basicamente em dois painéis: um para editar o código e o outro mostra o resultado renderizado na página. Isso significa que à medida que o código é editado, as mudanças são vistas em tempo real.

O CodePen permite que você escreva códigos sem efetuar nenhum cadastro. No entanto, se você quiser salvar seus projetos, deve criar uma conta. O acesso ao CodePen é através do link <http://codepen.io/>.

Uma outra alternativa é usar o terminal para efetuar a compilação. Para isso, é necessário instalar o Ruby e a gem do Sass. No Windows, a instalação do Ruby pode ser feita através do RubyInstaller, disponível em <http://rubyinstaller.org/>. Em sistemas operacionais como Linux e Mac OS, é necessário usar o sistema de gerenciamento de pacotes.

Mais informações podem ser obtidas em <https://www.ruby-lang.org/pt/documentation/installation/>.

Depois de ter o Ruby instalado, é necessário instalar o Sass via prompt de comando. A instalação do Sass é feita por meio do seguinte comando:

```
gem install sass
```

Em qualquer editor de texto — o Sublime (<https://www.sublimetext.com/>) é uma ótima opção —, você deve então criar o arquivo com a extensão `.scss` e pedir para que o Sass fique escutando o arquivo e traduzindo para CSS. O comando `watch`, apresentado a seguir, efetua a escuta do arquivo. Na linha de comando, é necessário o caminho do arquivo que possui as regras escritas com Sass e também o caminho do arquivo onde devem ser salvas as regras em CSS puro.

```
sass --watch style.scss:style.css
```

Agora se você pretende construir um projeto usando Sass e não está muito acostumado com o terminal, existem alternativas de softwares que possuem todo o processo de configuração e compilação através de interface gráfica. O Scout é um deles.

Com o Scout, você precisa configurar as pastas onde ficam o SCSS e o CSS gerado, e ele se encarrega da compilação. Também é possível efetuar algumas configurações como o estilo de saída que será abordado nos próximos tópicos. O download do Scout pode ser feito em <http://mhs.github.io/scout-app/>.

Em projetos maiores em que existe uma série de configurações tanto para o ambiente de desenvolvimento quanto para a produção, é possível utilizar uma ferramenta de automação de tarefas como o grunt (<http://gruntjs.com/>), ou o gulp (<http://gulpjs.com/>). Nestas é possível automatizar a varredura de arquivos `.scss`, configurar o estilo de saída, minificar e comprimir os arquivos para enviar ao servidor.

1.4 PROCESSO DE COMPILAÇÃO

Atualmente, os navegadores entendem apenas regras CSS para estilizar páginas. Quando escrevemos nossas folhas de estilo usando

Sass, estamos usando uma metalinguagem diferente do CSS nativo. Sendo assim, é necessária uma transformação dessas regras escritas para que os navegadores possam compreender. O Sass possui uma engine de compilação que transforma todas as regras escritas em CSS puro.



Figura 1.1: <https://goo.gl/p1ua9M>

O Sass possui duas sintaxes. A sintaxe original é usada em arquivos com extensão `.sass`. Essa sintaxe utiliza indentação para delimitar os blocos de estilo, e cada propriedade deve ser escrita em uma nova linha.

Sua grande desvantagem é a incompatibilidade com CSS nativo. No entanto, obriga os desenvolvedores deixarem o código indentado e, como não necessita de chaves e ponto e vírgula, o código fica mais limpo e muito mais legível. Programadores Ruby e Python, como já estão acostumados com sintaxe de indentação, normalmente preferem a sintaxe original, enquanto os demais optam pela sintaxe `.scss`. O trecho de código a seguir exibe a sintaxe `.sass`.

```
body
  background-color: gray
  font-family: Arial
  margin: 30px

.artigo
  background-color: white
  border-radius: 5px
  padding: 20px
  h2
```

```
color: #cf5c3f
```

A partir do Sass 3.0, é possível escrever arquivos Sass com a extensão `.scss`. No momento da escrita deste livro, a última versão estável do Sass é a 3.4.21. Esta possui sintaxe similar ao CSS nativo no qual os blocos de estilo são delimitados por chaves e as propriedades por ponto e vírgula.

A grande vantagem desta sintaxe é que a integração com os códigos escritos em CSS ocorre de forma natural, uma vez que a sintaxe é a mesma. No entanto, o código pode ficar confuso quando os desenvolvedores não costumam indentar as regras aninhadas. O código exibido a seguir possui as mesmas regras do trecho exibido anteriormente, no entanto, está usando a sintaxe `.scss`.

```
body{
  background-color: gray;
  font-family: Arial;
  line-height: 1.5;
  margin: 30px;

  .artigo{
    background-color: white;
    border-radius: 5px;
    padding: 20px;
    h2{
      color: #cf5c3f;
    }
  }
}
```

Neste livro, será usada a sintaxe `.scss`. Portanto, os arquivos que contém o código Sass devem possuir essa extensão. A opção pela sintaxe `.scss` foi pelo fato dela ser 100% compatível com o CSS nativo, ou seja, todo o CSS nativo é reconhecido pelo Sass. O mesmo não ocorre com arquivos de extensão `.sass`.

Após a compilação, o código CSS gerado deve ser similar ao que está sendo exibido a seguir, independente do tipo de extensão usada.

```
body {
```

```
background-color: gray;
font-family: Arial;
line-height: 1.5;
margin: 30px; }
body .artigo {
    background-color: white;
    border-radius: 5px;
    padding: 20px; }
    body .artigo h2 {
        color: #cf5c3f; }

/*# sourceMappingURL=style.css.map */
```

O resultado renderizado pelo browser pode ser conferido na figura seguinte.

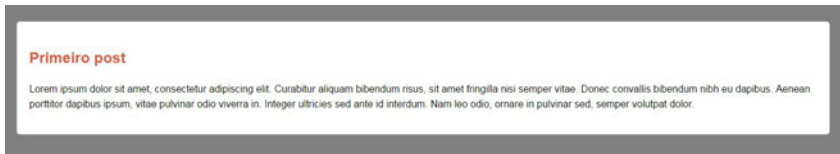


Figura 1.2: <https://goo.gl/JqL2hk>

É importante perceber que o código compilado possui a propriedade `sourceMappingURL`. Essa propriedade é responsável por indicar onde está o map do fonte original.

O mapa possui o nome do arquivo gerado, juntamente com a extensão `.map`. Este arquivo permite que o navegador consiga apontar para o trecho de código escrito que gera alguma regra de estilo. Veja na figura a seguir como o navegador Chrome exibe essa informação.

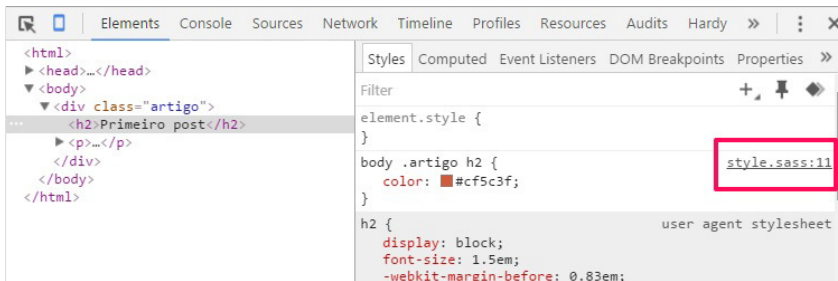


Figura 1.3: <https://goo.gl/8gKEfl>

Na pasta onde foi gerado o arquivo CSS, também é possível notar a presença de uma pasta chamada `.sass-cache`. Esta possui arquivos responsáveis pela aceleração da criação do CSS.

1.5 CONFIGURAÇÃO DE SAÍDA

O estilo do CSS resultante da compilação pode ser configurado. Temos algumas opções para escolher: *nested*, *expanded*, *compact* e *compressed*. Os próximos parágrafos descrevem cada um dos estilos de saída, juntamente com o comando para a geração de cada saída nos casos em que a linha de comando é usada para a compilação.

Nested

```
sass --watch style.scss:style.css --style nested
```

No `nested`, cada propriedade tem sua própria linha e as regras são recuadas com base no aninhamento do código Sass.

```
body {
  background-color: gray;
  font-family: Arial;
  line-height: 1.5;
  margin: 30px; }
body .artigo {
  background-color: white;
  border-radius: 5px;
  padding: 20px; }
body .artigo h2 {
```

```
color: #cf5c3f; }
```

Expanded

```
sass --watch style.scss:style.css --style expanded
```

Quando é usado o modo expandido, a saída fica similar à escrita de CSS nativo. As propriedades e regras ocupam uma linha, e propriedades são recuadas dentro das regras.

```
body {  
  background-color: gray;  
  font-family: Arial;  
  line-height: 1.5;  
  margin: 30px;  
}  
body .artigo {  
  background-color: white;  
  border-radius: 5px;  
  padding: 20px;  
}  
body .artigo h2 {  
  color: #cf5c3f;  
}
```

Compact

```
sass --watch style.scss:style.css --style compact
```

O modo compact coloca todas as propriedades de um seletor na mesma linha.

```
body { background-color: gray; font-family: Arial; line-height: 1.5;  
; margin: 30px; }  
body .artigo { background-color: white; border-radius: 5px; padding:  
: 20px; }  
body .artigo h2 { color: #cf5c3f; }
```

Compressed

```
sass --watch style.scss:style.css --style compressed
```

No modo comprimido todos os espaços e comentários são removidos, e o CSS fica todo na mesma linha. Esse é o estilo de saída

mais otimizado e deve ser utilizado quando sua página web estiver em produção.

```
body{background-color:gray;font-family:Arial;line-height:1.5;margin:30px}body .artigo{background-color:white;border-radius:5px;padding:20px}body .artigo h2{color:#cf5c3f}
```

Se você estiver usando um software como o Scout, ele possui as configurações de compilação em sua interface gráfica. Já em ferramentas de automação, como o gulp ou grunt, é possível passar a opção desejada na configuração.

1.6 USANDO COMENTÁRIOS

Comentários são fragmentos de texto que são ignorados pelo browser. Eles são usados para deixar o código mais inteligível. O Sass possui dois tipos de comentários:

- **Comentário de uma linha:** o uso de barra dupla (//) marca o início do comentário que se estende até o fim da linha.
- **Comentário de várias linhas:** os delimitadores /* e */ indicam início e fim de um comentário, e podem conter várias linhas. Este é padrão de comentário do CSS.

Vale ressaltar que a principal diferença entre os dois tipos de comentário existentes é que os comentários tradicionais do CSS (/* ... */) são compilados para o arquivo .css gerado. Já os comentários de linha única (//) não aparecem no CSS gerado.

```
/* Este comentário ocupa várias linhas e utiliza o formato padrão de comentário do CSS. Seu conteúdo aparece no arquivo CSS compilado. */
.titleInfo{
  background-color: #1EC800;
}
```



```
//Este formato de comentário pode ocupar somente uma linha.  
//Seu conteúdo não é compilado para o CSS gerado.  
.titleSuccess{  
  background-color: #00AAFF;  
}
```

1.7 O QUE ESPERAR DOS PRÓXIMOS CAPÍTULOS

Neste capítulo, foram elencados motivos para usar um pré-processador para escrever suas regras de estilo e o porquê da escolha pelo Sass. Também foi visto como ocorre o processo de compilação e algumas formas de configurar o ambiente de trabalho para começar a escrever regras de estilo com Sass.

É evidente que poucas funcionalidades foram exploradas. Os próximos capítulos tendem a revelar maneiras de aproveitar tudo o que o Sass fornece, como variáveis, aninhamento, mixins, herança, funções matemáticas, particionamento de arquivos e um pouco de lógica de programação.

REÚSE VALORES DE PROPRIEDADES COM VARIÁVEIS

Você já deve ter percebido que todas as aplicações web possuem um conjunto de padrões que permeiam todas as páginas. São padrões de cores, de tamanhos e de fontes. Esses padrões servem para dar uma identidade única à página.

Acontece que, de tempos em tempos, o time de design resolve dar uma nova roupagem à página e elabora um novo conjunto de padrões. Dentre as mudanças solicitadas, está a troca da paleta de cores. E claro, existem algumas cores que são utilizadas em vários locais e que precisam ser substituídas.

Em folhas de estilo construídas com CSS nativo, o método mais eficaz para efetuar as mudanças é o maravilhoso método arcaico de localizar e substituir, procurando pelo hexadecimal de um tom de azul que agora será substituído por um tom de verde. E claro, você utiliza esse método de varredura do código inúmeras vezes até efetuar todas as alterações sugeridas.

O Sass oferece uma maneira mais eficiente de gerenciar essas mudanças, usando variáveis. As variáveis consistem na forma fundamental e mais simples de reúso no Sass. Elas permitem que você nomeie valores de CSS que são usados repetidas vezes, e depois

referenciá-los pelo nome cada vez que você precisar.

Para desenvolvedores, o conceito de variáveis já soa natural. Já para os designers, as variáveis podem parecer intimidadoras a primeira vista.

Bom, uma variável pode armazenar o valor de uma cor, as propriedades de uma borda, um tamanho, uma família de fonte, enfim, tudo o que é atribuído como valor de uma propriedade CSS. O grande benefício do uso de variáveis é que quando for necessário efetuar uma mudança, esta será feita apenas na declaração da variável e propagada em todos os locais que a usam.

2.1 DECLARANDO UMA VARIÁVEL

O símbolo do dólar (\$) indica ao Sass que estamos definindo o início de uma variável. O nome da variável vem a seguir, sem espaços e pode conter letras, números, traços e underscores.

Após a declaração da variável, é necessário atribuir um valor a ela. A atribuição é feita da mesma forma que a atribuição de um valor a uma propriedade CSS: usando o símbolo de dois pontos (:), o valor e depois o símbolo de ponto e vírgula (;), como mostrado a seguir.

```
$nomeVariavel: valor;
```

Um uso típico de variáveis é para definir cores. Segue um exemplo da definição de uma variável que define a cor em que os links devem ser estilizados. Neste caso, o nome da variável é `corLink` e seu valor é o hexadecimal `#0096ED`.

```
$corLink: #0096ED;
```

As variáveis também podem ser usadas para declarar tamanhos, fontes, bordas e imagens. No trecho de código a seguir, podem ser


```
PSI1LjY4IiB5MT0iNS4yOTQiIHgyPSIyNS42MTciIHkyPSIyNS4yOTQiLz48bGluZS  
BmaWxsSPSJub25lIiBzdHJva2U9IiM4MDgwODAiIHNoZm9rZS13aWR0aD0iMiIgc3Ry  
b2tllW1pdGVybGltXQ9IjEwIiB4MT0iMjUuNjgiIHkxPSI1LjI5NCIgeDI9IjUuNz  
QzIiB5Mj0iMjUuMjk0Ii8+PHJlY3QgeD0iLTM2NC41IiB5PSItNDAiIGZpbGw9Im5v  
bmUiIHdpZHRoPSI0NDQiIGhlaWdodD0iMjgwIi8+PC9zdmc+);
```

USO DE IMAGENS SVG BASE 64

Existem muitos motivos para dar preferência ao uso de imagens no formato SVG, sendo o principal deles a independência de tamanho e resolução. Ao optar por usar uma imagem neste formato, é possível adicionar uma imagem à página de formas distintas.

Uma das maneiras existentes é através de Data URIs convertidos para a base 64. Uma Data URI é uma forma de codificar arquivos diretamente no CSS, sem necessidade de se fazer uma requisição HTTP.

A grande vantagem dessa abordagem é que a imagem fica embutida no CSS, evitando um request ao servidor. Além disso, se existir alguma mudança estrutural na sua página, não é necessário trocar o caminho do arquivo da imagem.

Propriedades CSS com múltiplos valores também podem ser declaradas como variáveis. Basta separá-los por espaços ou vírgulas, conforme o padrão da propriedade CSS. Segue um exemplo de definição de alguns valores para a propriedade `border` do CSS. A variável `bordaFocus` possui a dimensão, o estilo e a cor de uma borda. Esses valores estão separados por espaços.

```
$bordaFocus: 1px solid #0096ED;  
$fontePrincipal: "Trebuchet MS", Verdana, Arial, serif;
```

Uma família de fonte também pode ser declarada através de

uma variável. Vale lembrar de que é uma prática comum declarar uma lista de fontes ordenadas conforme a prioridade de apresentação na página, já que nem todos os dispositivos que podem acessar a página possuem a imensa quantidade de fontes disponíveis. É altamente recomendável que a lista de fontes termine com uma fonte genérica.

No exemplo a seguir, foi definida uma variável com o nome `courier` que contém uma lista de fontes separadas por vírgula.

```
$courier: "courier new", courier, freemono, "nimbus mono l", "liberation mono", monospace;
```

2.2 REFERENCIE UMA VARIÁVEL

Além de definir uma variável, é importante usá-la para aproveitar seus benefícios. Para usar uma variável, basta você colocar o seu nome no local onde normalmente seria usado o valor de uma propriedade.

```
$corSecundaria: #F2F2F2;

.publicidade{
    background-color: $corSecundaria;
}

.artigo{
    color: $corSecundaria;
}
```

No trecho de código apresentado, foi declarada a variável `corSecundaria`. Ela está sendo usada por duas seções distintas da página. Na seção `publicidade`, é usada como cor de fundo e, na seção `artigo`, é usada como cor da fonte.

No momento da compilação, os locais onde usam esta variável são substituídos pelo hexadecimal definido na variável. A seguir, está o mesmo trecho de código compilado.

```
.publicidade {
  background-color: #F2F2F2; }

.artigo {
  color: #F2F2F2; }
```

Vamos supor que o time de design resolva trocar essa cor. A única modificação a ser feita é no valor da variável. O restante do código continuará exatamente o mesmo. No momento da compilação, o valor será substituído em todos os locais que usam a variável `corSecundaria`. Acompanhe a seguir:

```
$corSecundaria: #B3B3B3;

.publicidade{
  background-color: $corSecundaria;
}

.artigo{
  color: $corSecundaria;
}
```

Já vimos que uma variável pode substituir o valor de uma propriedade CSS. Também podemos usar uma variável na declaração de outra variável. Veja o exemplo:

```
$corFocus: #0096ED;
$bordaFocus: 1px $corFocus solid;
```

Vale lembrar de que apenas as variáveis usadas são compiladas. Temos no exemplo a seguir duas variáveis declaradas: `corAlerta` e `tamanhoMargem`. Note que, no trecho apresentado, apenas a variável `tamanhoMargem` está sendo referenciada. A variável `corAlerta` está declarada, mas não está sendo usada.

```
$corAlerta: #FACD52;
$tamanhoMargem: 1%;

body{
  margin: $tamanhoMargem;
}
```

Este trecho compilado deve ficar como o CSS mostrado a seguir.

O valor `#FACD52` não aparece no CSS compilado, já que a variável que o representa não foi usada.

```
body {  
  margin: 1%; }
```

É importante ressaltar que uma variável só pode ser usada após a sua declaração. Se existir a tentativa de utilizar uma variável que ainda não foi declarada, o compilador do Sass emitirá uma mensagem de erro: *Syntax error: Undefined variable: "\$iconeFechar"*.

Por este motivo, é uma boa prática definirmos as variáveis no topo do arquivo. Também é possível criar um arquivo para definir as variáveis. Isto será abordado nos capítulos seguintes.

2.3 VARIÁVEIS GLOBAIS E DE ESCOPO

No Sass, as variáveis podem ser globais ou de escopo. As variáveis globais podem ser utilizadas em toda a folha de estilo. Elas devem ser declaradas no topo do arquivo. Todos os exemplos mostrados neste capítulo até o momento compreendem exemplos de variáveis globais.

Já as variáveis de escopo aparecem declaradas dentro de um seletor, e só podem ser utilizadas dentro dele. Se você tentar usar uma variável de escopo fora do seletor onde ela foi declarada, o compilador do Sass não efetuará a compilação da folha de estilo e deve emitir um erro de variável indefinida.

No trecho de código a seguir, temos a variável `corAlerta` que está declarada dentro do seletor `botaoAlerta`. Ela só pode ser usada dentro deste seletor.

```
.botaoAlerta{  
  $corAlerta: #FACD52;  
  background-color: $corAlerta;  
}
```


2.4 COMO NOMEAR VARIÁVEIS

Nomear variáveis nunca é uma tarefa fácil. Ao dar um nome para uma variável, é importante que este revele o seu propósito. Ou seja, ao ler o nome da variável, é importante ter ideia do conteúdo que ela está armazenando.

Além disso, é importante estar atento às convenções utilizadas pela comunidade de desenvolvimento, e elencar padrões que você e sua equipe devem seguir. Vale ressaltar que o Sass adota algumas regras no momento da compilação, e estas estão descritas nos parágrafos seguintes.

Um nome composto pode ser separado por traços, underscores, ou podemos ainda usar o padrão *camelCase*. No CSS, é mais comum o uso de traços para separar nomes compostos. Eu, particularmente, prefiro utilizar *camelCase*. Você pode optar por qualquer um dos estilos. No entanto, é essencial definir um padrão para a nomeação de todas as variáveis e seletores.

É importante ressaltar que o Sass diferencia letras maiúsculas e minúsculas nos nomes de variáveis. Dessa forma, você não pode declarar uma variável como `$corAlerta` e tentar utilizá-la como `$coralerta`. No entanto, se você usar traço para separar um nome composto, poderá referenciar a variável com underscore e vice-versa.

Sendo assim, você pode declarar uma variável como `$cor-alerta` e referenciá-la como `$cor_alerta`. Vale lembrar de que isso não é uma boa prática, embora o compilador não reclame.

2.5 FAÇA MUITO MAIS COM VARIÁVEIS

As variáveis representam um recurso muito poderoso do Sass e é importante que você consiga explorá-las ao máximo. Seguem dois

exemplos de como tirar um bom proveito com o uso de variáveis.

Crie uma paleta de cores com Sass

É bem provável que sua aplicação possua uma paleta de cores definida pela equipe de design. Se não existir, é muito importante definir cores que podem ser usadas em suas páginas, para que exista uma consistência entre elas.

É uma excelente prática transformar sua paleta de cores em variáveis. Isso facilita bastante a transcrição das especificações feitas pela equipe de design para regras CSS. No trecho de código a seguir, estão definidas algumas cores que fazem parte da paleta de uma aplicação web.

```
$corFundoDestaque: #deedf7;  
$corBordaDestaque: #aed0ea;  
$corTextoDestaque: #222222;  
$corFundoConteudo: #f2f5f7;  
$corBordaConteudo: #dddddd;  
$corFundoHover: #e4f1fb;  
$corBordaHover: #74b2e2;  
$corTextoHover: #0070a3;  
$corFundoAtivo: #3baae3;  
$corBordaAtivo: #2694e8;  
$corTextoAtivo: #ffffff;  
$corFundoErro: #cd0a0a;  
$corTextoErro: #ffffff;  
$corSombra: #cccccc;
```

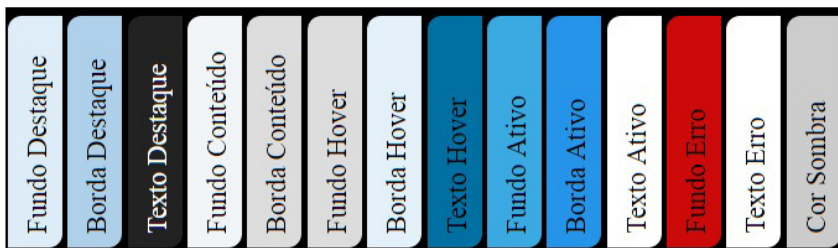


Figura 2.1: <https://goo.gl/jCIOEM>

Crie uma biblioteca de famílias de fontes

Já foi mencionado que uma lista de fontes pode ser definida em uma variável para que não seja necessário copiar e colar essa lista em todos os locais em que a fonte deve ser usada. Porém, sua aplicação web deve possuir mais de um tipo de fonte. E se você produz folhas de estilo para diversas aplicações web, é muito provável que você tenha inúmeras listas de famílias de fontes.

Com o Sass, você pode criar uma biblioteca contendo essas listas. Como o compilador inclui apenas as variáveis utilizadas, você terá suas fontes sempre a mão e não precisa se preocupar com o tamanho do CSS gerado.

Segue um exemplo de biblioteca de família de fontes. Nos capítulos posteriores, será mostrado como separá-la em um arquivo distinto.

```
$lucida: "lucida grande", "lucida sans unicode", "lucida sans", lucida, sans-serif;
$helvetica: "helvetica neue", arial, helvetica, freesans, "liberation sans", "numbus sans l", sans-serif;
$geneva: geneva, tahoma, "dejavu sans condensed", sans-serif;
$courier: "courier new", courier, freemono, "nimbus mono l", "liberation mono", monospace;
$verdana: verdana, "bitstream vera sans", "dejavu sans", "liberation sans", geneva, sans-serif;
$cambria: cambria, georgia, "bitstream charter", "century schoolbook l", "liberation serif", times, serif;
$times: times, "times new roman", "nimbus roman no9 l", freeserif, "liberation serif", serif;
```

2.6 UM PROJETO REAL

Quando estamos aprendendo uma tecnologia nova, é muito comum não conseguirmos enxergar sua aplicabilidade em um projeto real. Pensando nisso, a partir deste capítulo, nós usaremos as funcionalidades já exploradas do Sass para construir uma página visualmente muito similar ao Facebook.

Neste momento, vamos montar uma base estrutural e aplicar o

uso das variáveis na folha de estilo. No decorrer dos próximos capítulos, incrementaremos a página com as funcionalidades do Sass apresentadas.

Para montar a estrutura da página, neste primeiro momento, vamos usar apenas HTML e CSS. A primeira versão da página deve ficar similar à figura seguinte.

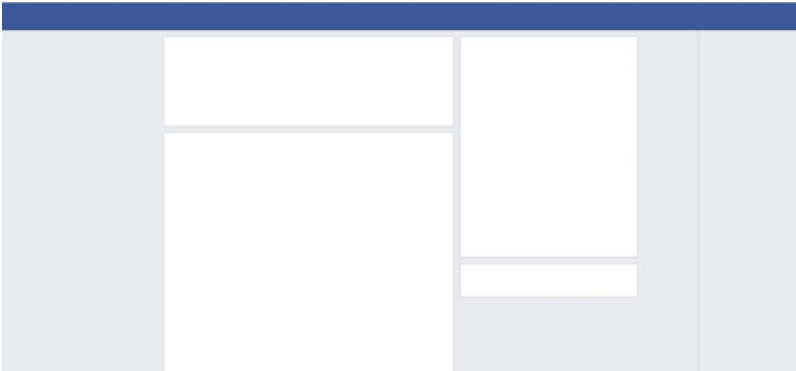


Figura 2.2: <https://goo.gl/wXsVTC>

Uma análise preliminar indica a ocorrência de alguns blocos na página. Estes blocos são representados como `div` s no HTML. Veja no trecho HTML a seguir a estrutura de `div` s que foi montada para a construção da página. Uma hierarquia de `div` s foi criada para que o conteúdo se apresentasse da forma adequada.

```
<body>
  <div class="cabecalho">
  </div>
  <div class="conteudo">
    <div class="conteudoPrincipal">
      <div>
        <div class="informacoesConta">
        </div>
        <div class="feedNoticias">
          <div class="boxInicial"></div>
          <div class="boxPublicacao"></div>
          <div class="boxPublicacao"></div>
        </div>
      </div>
    </div>
  </div>
```

```

        <div class="outrasInformacoes">
            <div class="boxInfo"></div>
            <div class="boxIdiomas"></div>
        </div>
    </div>
    <div class="sideBar"></div>
</div>
</body>

```

Como já deve ser de seu conhecimento, apenas com marcação HTML, não é possível montar uma página esteticamente atraente. Precisamos de CSS. Neste primeiro momento, o nosso esforço se concentrará na diagramação do conteúdo.

Toda a página deve ser iniciada com a limpeza de algumas formatações que são aplicadas pelo browser. Vamos zerar as propriedades de `margin` e `padding`, e aplicar uma cor de fundo. Acompanhe o trecho de código seguinte.

```

body{
    margin: 0;
    padding: 0;
    background-color: #E9EBEE;
}

```

Pela imagem, é possível perceber que o cabeçalho possui uma cor de fundo diferente. Vamos aplicá-la, juntamente com a definição de altura e de uma borda.

```

.cabecalho{
    height: 45px;
    background-color: #3B5998;
    border-bottom: 1px solid #29487D;
}

```

O passo seguinte é começar a montar o conteúdo. Neste momento, basicamente são definidos os tamanhos e a alocação dos blocos de conteúdo, como mostram as estilizações seguintes.

```

.conteudo{
    min-height: 600px;
    display: flex;
    justify-content: space-between;
}

```

```

}
.conteudoPrincipal{
    display: flex;
    justify-content: center;
    width: 100%;
}

.conteudoPrincipal > div{
    display: flex;
    margin: 10px;
}

[class*="box"]{
    background-color: #FFFFFF;
    border-radius: 3px;
    border: 1px solid #CCC;
    margin: 0px 10px 10px 0;
}
.sideBar{
    width: 200px;
    border-left: 1px solid #CCC;
}

.informacoesConta{width: 180px;}
.feedNoticias{width: 500px;}
.outrasInformacoes{width: 310px;}
.boxInicial{height: 150px;}
.boxPublicacao{height: 500px;}
.boxInfo{height: 370px;}
.boxIdiomas{height: 55px;}

```

Note que, inicialmente, colocamos uma altura mínima para a nossa `div conteudo` que representa toda a página, exceto o cabeçalho. Nela também adicionamos a propriedade `flex`, setando o ajuste de conteúdo como `space-between`, que aloca espaço entre os itens presentes dentro do container, se estes não ocuparem o tamanho máximo. Isso significa que como esta `div` possui dois filhos que representam o conteúdo principal e o sidebar, se houver espaço sobrando na tela, ele ficará entre essas duas `divs`.

O conteúdo principal alinha seus filhos de uma forma diferente. Note que é usado o ajuste de conteúdo `center`. Neste, os filhos ficam juntos e alinhados ao centro. Quanto as demais estilizações

dispensam comentários, pois são apenas definições de tamanho e demarcações de bordas que você já deve estar acostumado.

Com esses trechos de CSS, você terá uma página similar à imagem apresentada no início do tópico. Note que, até o momento, a página foi construída apenas com CSS.

Em uma análise superficial, podemos vislumbrar alguns valores de propriedades que podem ser definidas como variáveis para facilitar nossa futura manutenção. Veja:

```
$corBase: #3B5998;
$corFontePrincipal: #1D2129;
$corFonteDestaque:#FFFFFF;
$corFonteLink: #365899;
$corFundo: #E9EBEE;
$corBorda: #CCC;
$corFundoBoxes: #FFFFFF;
$tamanhoFonte: 12px;
$familiaFonte: helvetica,arial,sans-serif;
```

Definimos uma paleta inicial de cores e também variáveis que controlam o estilo da fonte. Após a definição, podemos fazer o uso delas em nossos blocos de estilo. Os blocos de código seguintes mostram a substituição:

```
body{
    margin: 0;
    padding: 0;
    background-color: $corFundo;
}

.cabecalho{
    height: 45px;
    background-color: $corBase;
    border-bottom: 1px solid #29487D;
}

[class*="box"]{
    background-color: $corFundoBoxes;
    border-radius: 3px;
    border: 1px solid $corBorda;
    margin: 0px 10px 10px 0;
}
```

```
.sideBar{
  width: 200px;
  border-left: 1px solid $corBorda;
}
```

Note que nem todas as variáveis definidas foram usadas neste momento. Faremos o uso delas nos próximos capítulos, nos quais os blocos de conteúdo serão estilizados. Também é muito provável que novas variáveis sejam definidas no transcorrer da construção das estilizações.

Se você observar atentamente os blocos de estilo construídos, vai perceber que existem algumas cores que são usadas e que não estão definidas como variáveis. Outras propriedades como tamanhos, estilos de borda também não estão.

É importante que você entenda que variáveis devem ser declaradas apenas quando existe o intuito de usá-las em mais de um momento em sua folha de estilo. Não cometa o erro de declarar todos os valores de propriedades como variáveis.

2.7 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Neste capítulo, foi apresentado o primeiro recurso do Sass que permite o reúso em folhas de estilo: as variáveis. Foi visto como declarar uma variável, como referenciá-la e onde é indicado o seu uso. Também foram apresentadas as diferenças entre variáveis globais e de escopo, e algumas regras e convenções para a sua nomeação.

Neste capítulo, também iniciamos a construção de uma página, usando as funcionalidades presentes no Sass. Essa página será aprimorada a cada capítulo. O próximo abordará o recurso de aninhamento, que consiste em uma forma de aninhar regras de estilo quando existe a necessidade de usar um conjunto ou

concatenação de seletores.

ESCREVA SELETORES LONGOS MAIS RÁPIDO COM ANINHAMENTO

A filosofia por trás dos pré-processadores de CSS consiste na redução de repetição de código nas folhas de estilo. Quem trabalha com CSS sabe o quão oneroso é escrever regras de estilo que se aplicam para apenas uma seção da página, mas que contêm elementos presentes em seu todo.

O Sass possui um mecanismo que facilita a construção de regras de estilo quando é necessário aplicá-las para uma hierarquia de seletores. Esse mecanismo é chamado de aninhamento.

3.1 COMO ANINHAR REGRAS DE ESTILO

Não é nenhuma novidade que o HTML possui uma estrutura hierárquica. Com o CSS, não é possível manter a mesma hierarquia do HTML. Para aplicar as regras de estilo, é necessário escrever o mesmo seletor diversas vezes. Veja o exemplo:

```
<nav class="menuSuperior">
  <ul>
    <li><a href="#lançamento">Lançamento</a></li>
    <li><a href="#marca">Marca</a></li>
    <li><a href="#campanha">Campanha</a></li>
    <li><a href="#colecão">Coleção</a></li>
    <li><a href="#contato">Contato</a></li>
```

```
</ul>  
</nav>
```

O trecho de código apresentado anteriormente representa o menu principal de uma página web. Nele, é possível identificar elementos como `<ul>`, `<li>` e `<a>`. Estes não são elementos específicos do menu principal e podem ser usados em outras partes da página. Porém, é desejável que algumas regras de estilo sejam aplicadas para esses elementos apenas quando estão no menu principal.

Usando CSS nativo, a estilização do menu deve ficar similar ao trecho de código exibido a seguir. Note que existem muitas repetições de seletores. Além disso, se for necessário trocar a class do elemento `nav`, várias alterações devem ser feitas no CSS para que a estilização permaneça intacta.

```
.menuSuperior { ... }  
.menuSuperior ul {...}  
.menuSuperior ul li {...}  
.menuSuperior ul li a {...}
```

O Sass permite que as regras sejam aninhadas, evitando a escrita duplicada de seletores. Acompanhe a seguir a estilização do menu usando Sass.

```
.menuSuperior{  
  padding-top: 40px;  
  ul {  
    margin:0;  
    padding:0;  
    list-style-type: none;  
    text-align: center;  
    li {  
      display: inline;  
      a {  
        text-decoration: none;  
        padding: 10px 30px;  
        color: #FFFFFF;  
        background-color: #1C1C1C;  
        font-size:18px;  
      }  
    }  
  }  
}
```

```

    }
  }
}

```

No trecho de código exibido anteriormente, é possível observar que as regras de estilo seguem uma estrutura hierárquica muito similar ao HTML que deve ser estilizado. Isso significa que o Sass permite que regras sejam colocadas dentro de regras. Para isso, é necessário colocar um novo seletor antes do fechamento das chaves e inserir novas regras. Este mecanismo pode ser aplicado quantas vezes forem necessárias.

Na estilização do menu, é possível perceber que inicialmente é aplicado um `padding` para a classe `.menuSuperior`. Dentro do seletor de classe, foi inserido um novo seletor `ul`, e são aplicadas regras de estilo para ele. Isso acontece também para o seletor `li` e `a`.

Após a compilação, o CSS gerado deve ficar similar ao trecho de código exibido a seguir. Vale lembrar que é possível configurar o tipo de saída como foi mencionado no capítulo 1.

```

.menuSuperior {
  padding-top: 40px; }
.menuSuperior ul {
  margin: 0;
  padding: 0;
  list-style-type: none;
  text-align: center; }
.menuSuperior ul li {
  display: inline; }
.menuSuperior ul li a {
  text-decoration: none;
  text-decoration: none;
  padding: 10px 30px;
  color: #FFFFFF;
  background-color: #1C1C1C;
  font-size: 18px; }

```

Note que existe repetição de alguns seletores, pois o CSS nativo não tem suporte a indentação. No entanto, como a manutenção é

dada no arquivo `scss` , o que nos interessa realmente são as possibilidades que o Sass oferece.

Usando o HTML e as regras escritas em Sass exibidas anteriormente, o menu estilizado deve ficar similar ao da figura a seguir.



Figura 3.1: <https://goo.gl/X2eBNJ>

3.2 USANDO O & PARA CONCATENAR

Normalmente, no momento da compilação de regras aninhadas, o Sass conecta os seletores usando um espaço em branco entre eles. No entanto, em alguns momentos, é necessário um comportamento ligeiramente diferente.

Para que a conexão de uma regra aninhada seja feita sem o uso do espaço em branco, é necessário usar o símbolo `&` . Existem alguns casos bem comuns nos quais o `&` se torna necessário. Vamos explorar alguns nos parágrafos seguintes.

No menu estilizado anteriormente, adicionaremos regras de estilo para diferenciar um item de menu quando o mouse está posicionado sobre ele. Para conseguir este efeito, é necessário usar o pseudosseletor `hover` .

O trecho de código a seguir mostra como conseguir concatenar o pseudosseletor usando aninhamento. Note que as regras de estilo mencionadas anteriormente foram suprimidas, e foram substituídas por `...` para focar no conceito apresentado neste tópico.

```
.menuSuperior{  
  ...  
  ul {  
    ...
```

```

li {
  ...
  a {
    ...
    &:hover {
      background-color: #999999;
      color: #000000;
      box-shadow: 0 5px 10px #CCC;
      text-shadow: 1px 1px 1px #CCC;
    }
  }
}

```

No código apresentado, note o símbolo `&` sendo usado antes do pseudosseletor `hover`. Ele se faz necessário porque a sintaxe do pseudosseletor é `a:hover`. Usando apenas o aninhamento de seletores, a compilação coloca um espaço antes de considerar o próximo item.

Ou seja, usando aninhamento comum, o código gerado seria `a :hover` e o estilo não seria aplicado corretamente. O CSS exibido a seguir, representa o código descrito compilado.

```

.menuSuperior {
  ...}
.menuSuperior ul {
  ...}
.menuSuperior ul li {
  ...}
.menuSuperior ul li a {
  ... }
.menuSuperior ul li a:hover {
  background-color: #999999;
  color: #000000;
  box-shadow: 0 5px 10px #CCC;
  text-shadow: 1px 1px 1px #CCC; }

```

Muitas vezes, precisamos da combinação de seletor juntamente com uma classe CSS, ou o agrupamento de várias classes no mesmo seletor, para que um estilo seja aplicado. Para exemplificar, vamos pensar em um formulário que possui um campo de seleção que

apresenta uma borda vermelha do lado esquerdo quando é obrigatório. E quando o usuário passar por ele e nenhuma opção for selecionada, deve apresentar a borda em todos os lados, alertando-o de que existe algo errado no campo.

Geralmente para conseguir esse efeito, adicionamos classes CSS dinamicamente com a ajuda de alguma linguagem de programação. No exemplo apresentado a seguir, é adicionada a classe `invalido` no seletor, após o usuário sair do campo se nenhum valor foi selecionado.

```
<select class="obrigatorio invalido">
  <option value="">Selecione</option>
  <option value="valor1">Valor 1</option>
  <option value="valor2">Valor 2</option>
</select>
```

É importante observar que a borda deve aparecer em todos os lados apenas se o elemento `select` for obrigatório e estiver com a classe `invalido`. Para conseguir aplicar o estilo apenas quando as duas condições forem satisfeitas, usamos as duas classes CSS. No Sass, para conseguir isso, também usamos o símbolo `&`. Temos a seguir o código escrito com Sass.

```
$colorRequired: #FF0000;
.obrigatorio{
  box-shadow: -2px 0px 0px 0px $colorRequired;

  &.invalido{
    border: 1px solid $colorRequired;
  }
}
```

No CSS compilado, exibido a seguir, é importante perceber a falta de espaço entre as classes `.obrigatorio` e `.invalido`. Isso é vital para que o estilo seja aplicado corretamente.

```
.obrigatorio {
  box-shadow: -2px 0px 0px 0px #FF0000; }
.obrigatorio.invalido {
  border: 1px solid #FF0000; }
```

A renderização do estilo pelo browser deve ficar similar a:



Figura 3.2: <https://goo.gl/cgvZob>

Um outro contexto em que o símbolo `&` pode ser usado é na composição de nomes de classes CSS. O HTML apresentado a seguir representa um card composto por um título e um conteúdo, que possuem como nomes de classes `card-titulo` e `card-conteudo`.

```
<div class="card">
  <div class="card-titulo">
    <h1>Sass</h1>
  </div>
  <div class="card-conteudo">
    <p>... </p>

    <p>... </p>
  </div>
</div>
```

O código Sass apresentado mostra como fazer a composição de nomes de classe. Note que `-titulo` e `-conteudo` são prefixados pelo símbolo `&`.

```
.card{
  background: white;
  box-shadow: 0 5px 15px 0 rgba(0,0,0,0.25);
  width:400px;

  &-titulo{
    padding: 5px;
    color: white;
    ...
  }

  &-conteudo{
    color: #777;
    padding: 20px;
    line-height: 1.5;
  }
}
```



```
}
```

O resultado da estrutura do CSS gerado é exibido a seguir, juntamente com o resultado renderizado pelo browser.

```
.card {  
  ... }  
.card-titulo {  
  ... }  
.card-conteudo {  
  ... }
```



Figura 3.3: <https://goo.gl/PnJz7s>

3.3 ANINHAMENTO DE PROPRIEDADES

Além do aninhamento de seletores, com Sass também é possível aninhar propriedades que possuem o nome composto. Para isso, é

necessário colocar o primeiro nome da propriedade e adicionar o símbolo de dois pontos antes da chave de abertura.

Essa funcionalidade é interessante quando precisamos setar vários valores de propriedades que iniciem com o mesmo nome. Como `font-family`, `font-size`, `font-weight` ou `text-align`, `text-indent` ou ainda `border-top`, `border-bottom`. No exemplo a seguir, estas propriedades estão declaradas usando aninhamento.

```
span{
  text:{
    align:center;
    indent: 20px;
  }

  font:{
    family: 'Helvetica';
    size: 30px;
  }

  border:{
    top:{
      style:dashed;
      left:{
        radius: 10px;
      }
    }
  }
}
```

Note que a propriedade `border` possui dois níveis de aninhamento. Isso também é possível se a propriedade possuir mais de duas palavras. A compilação deve ficar como o CSS mostrado adiante.

```
span {
  text-align: center;
  text-indent: 20px;
  font-family: 'Helvetica';
  font-size: 30px;
  border-top-style: dashed;
  border-top-left-radius: 10px; }
```

Em propriedades que possuem uma forma de escrita curta como a propriedade `padding`, no qual os valores podem ser colocados na mesma propriedade, é possível efetuar cálculos usando o aninhamento. No trecho de código apresentado em seguida, temos um valor de `padding` para a esquerda, e o `padding` da parte superior deve ser correspondente à metade do `padding` esquerdo.

```
span{
  $padding: 20px;
  padding: {
    left: $padding;
    bottom: $padding / 2;
  }
}
```

A compilação do código anterior deve ficar similar ao CSS que segue.

```
span {
  padding-left: 20px;
  padding-bottom: 10px;}
```

Você pode usar esse recurso também para escrever as exceções. Imagine que a sua caixa de texto deve possuir a borda superior na cor azul e as demais na cor cinza. Você pode declarar o comportamento padrão e aninhar a propriedade que é específica. Veja:

```
input[type="text"]{
  border: 1px solid #ccc {
    top-color: #0000FF;
  }
}
```

A compilação deve ser como mostrado a seguir:

```
input[type="text"] {
  border: 1px solid #ccc;
  border-top-color: #0000FF; }
```

A figura na sequência exibe a caixa de texto com o estilo aplicado.



Figura 3.4: <https://goo.gl/tQFLqs>

3.4 ANINHAMENTO DE GRUPOS DE SELETORES

O Sass permite o aninhamento de grupo de seletores. Isso é extremamente útil quando existem vários níveis de seletores e você precisa aplicar um estilo para um determinado grupo.

Note no exemplo a seguir que temos no primeiro aninhamento o seletor `.titulo` e, dentro dele, `h1`, `h2` e `h3` compartilham a mesma regra. No segundo aninhamento, temos os seletores `.titulo` e `.informacoes` e, para estes dois, temos uma regra que deve ser aplicada para o `h3`.

```
.titulo{
  h1, h2, h3{
    margin-left: 20px;
  }
}

.titulo, .informacoes{
  h3{
    color: #0000FF;
  }
}
```

No código seguinte, é possível verificar a compilação das regras.

```
.titulo h1, .titulo h2, .titulo h3 {
  margin-left: 20px; }

.titulo h3, .informacoes h3 {
  color: #0000FF; }
```

3.5 COMBINAÇÃO DE FILHOS E IRMÃOS (>, + E ~)

A combinação de seletores CSS permite a construção de estilizações mais robustas. A mais comum é a combinação de seletores descendentes em que um espaço é usado entre os seletores.

Esse tipo de combinação já foi explorado nos tópicos anteriores. O CSS permite a inserção de algumas regras de combinações entre filhos e irmãos, e este tópico falará sobre elas, usando Sass.

O símbolo `~` (til) é chamado de seletor irmão geral e estiliza o segundo elemento apenas se for precedido pelo primeiro. Imagine que você possua conteúdo separado em parágrafos e quer adicionar um estilo diferente aos parágrafos que seguem um `<h2>` como no HTML apresentado.

```
<div class="conteudo">
  <p>O Sass começou a ser criado em 2006.</p>
  <h2>Mais informações...</h2>
  <p>O processador do Sass é escrito em Ruby.</p>
  <p>Sass é open source.</p>
</div>
```

O caso citado possui uma `div` conteúdo, que tem vários filhos. Precisamos estilizar os parágrafos que sucedem o elemento `h2`. O símbolo `~` faz exatamente isso. O código Sass exibido a seguir demonstra como efetuar essa estilização.

```
.conteudo{
  h2{
    font-family: 'Arial';
    color: dimgrey;
    ~p{
      font-size: 18px;
      color: #999999;
    }
  }
}
```

Após a compilação, o CSS gerado deve ser o seguinte:

```
.conteudo h2 {
  font-family: 'Arial';
  color: dimgrey; }
```

```
.conteudo h2 ~ p {  
  font-size: 18px;  
  color: #999999; }
```

A renderização no browser deve ficar similar à figura a seguir. Note que o primeiro parágrafo não foi estilizado, já que ele não possui o elemento `h2` como antecessor.

O Sass começou a ser criado em 2006.

Mais informações ...

O processador do Sass é escrito em Ruby.

Sass é open source.

Figura 3.5: <https://goo.gl/PjLrGP>

O símbolo `+` (mais) é chamado de seletor irmão adjacente, e só estiliza o segundo elemento se este vier imediatamente após o primeiro elemento. Um contexto de uso muito comum para esse tipo de combinação é a divisão de um painel em diversas colunas com uma margem entre elas. Vale ressaltar que a margem só pode existir entre as colunas.

Uma forma elegante de fazer isso é aplicando uma margem ao lado esquerdo sempre que uma coluna vier imediatamente após outra coluna. Isso significa que a primeira coluna não possui nenhuma coluna antes dela, o que quer dizer que ela não receberá o estilo. As demais colunas possuem uma coluna antes e o estilo vai ser aplicado.

Note no trecho de código a seguir a utilização do símbolo `&`, indicando que será usado o seletor do nível anterior e o símbolo `+`, que é o seletor adjacente.

```
.col{
  width: 200px;
  height: 200px;
  float:left;
  background: #00FF00;
  &+&{
    margin-left: 20px;
  }
}
```

A compilação deve gerar o CSS mostrado adiante. Perceba nas últimas duas linhas que a `margin-left` só deve ser aplicada em uma coluna que possui como irmão anterior uma outra coluna.

```
.col {
  width: 200px;
  height: 200px;
  float: left;
  background: #00FF00; }
.col + .col {
  margin-left: 20px; }
```

Isso pode ser conferido na figura que representa a renderização feita pelo browser. É importante perceber que não foi aplicado margem nas extremidades da primeira e última coluna, apenas entre elas.

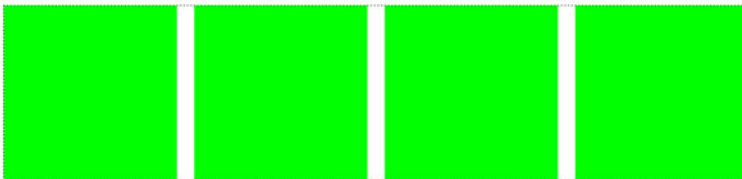


Figura 3.6: <https://goo.gl/UZhgbB>

O símbolo `>` é chamado de seletor filho direto, e seleciona somente os filhos diretos do primeiro seletor. Imagine que você possui o HTML descrito a seguir e precisa estilizar a página de forma que cada tópico do conteúdo fique com aparência de boxes na página.

```

<div class="conteudo">
  <div>
    <h3>Combinação de filhos e irmãos: >, +, e ~</h3>
    <div><p>O símbolo ~ é chamado de seletor irmão geral e est
iliza o segundo elemento apenas se for precedido pelo primeiro.</p>
</div>
    <div><p>O símbolo + é chamado de seletor irmão adjacente e
só estiliza o segundo elemento se este vier imediatamente após o
primeiro elemento</p></div>
    <div><p>O símbolo > é chamado de seletor filho direto e se
leciona somente os filhos diretos do primeiro seletor.</p></div>
  </div>
  <div>
    <h3>Usando o & para concatenar</h3>
    <div><p>Normalmente, no momento da compilação de regras an
inhadas, o Sass conecta os seletores usando um espaço em branco en
tre eles. No entanto, em alguns momentos, é necessário um comporta
mento ligeiramente diferente. Para que a conexão de uma regra anin
hada seja feita, sem o uso do espaço em branco, é necessário usar
o símbolo "&"</p></div>
  </div>
</div>

```

Cada tópico da página está dentro de uma div que contém outros elementos, inclusive outras divs. É possível nomear as divs para conseguir o efeito. Uma outra solução possível é usar o seletor filho direto, como mostrado no trecho de código a seguir:

```

.conteudo {
  background: black;
  padding: 10px;
  > div {
    background: beige;
    border-radius: 10px;
    padding: 10px;
    margin: 10px;
  }
}

```

Note o uso do símbolo > . Ele indica que apenas as divs filhas de conteúdo serão estilizadas. Isso pode ser conferido na imagem de renderização do browser:

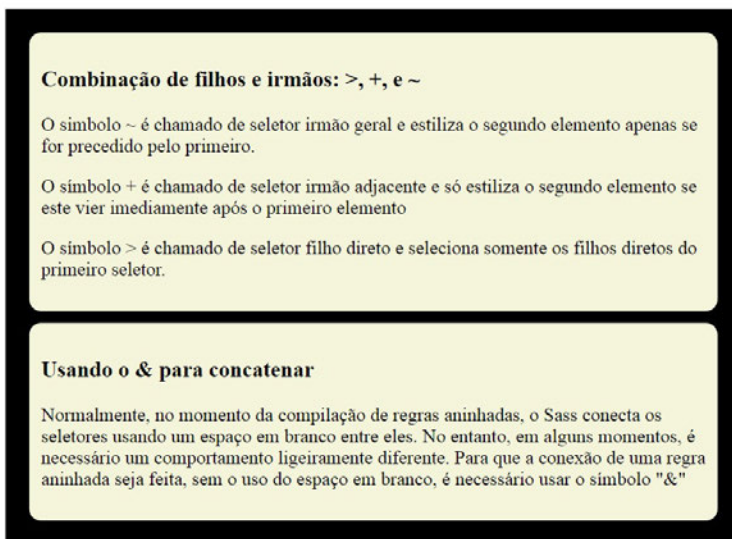


Figura 3.7: <https://goo.gl/v6l0PM>

3.6 PERIGOS DO ANINHAMENTO EXCESSIVO

O recurso de aninhamento para CSS é maravilhoso. No entanto, é importante se atentar para não o usar incorretamente. Segue um trecho de HTML — não muito organizado, mas que já vi similares em aplicações comerciais — para exemplificar.

```
body>
  <div>
    <div class="container">
      <div class="artigo">
        <div class="titulo">
          <h1>...</h1>
          <h2>...</h2>
        </div>
        <div class="conteudo">
          <div class="secaoPrincipal">
            <div class="redes">
              <nav>
                <ul>
                  <li><a href="#"></a></li>
                  <li><a href="#"></a></li>
                  <li><a href="#"></a></li>
```

```

        </ul>
      </nav>
    </div>
  </div>
</div>
</div>
</div>
</div>
</body>

```

Para a estilização do HTML mencionado, é possível usar o aninhamento que o Sass disponibiliza. Veja:

```

body{
  .container{
    .artigo{
      .titulo{
        h1{
          font-size: 20px;
        }
        h2{
          font-size: 15px;
        }
      }
      .conteudo{
        background-color: #CCC;
        .secaoPrincipal{
          font-size: 14px;
          .redes{
            padding: 10px;
            nav{
              padding-top: 40px;
              ul{
                margin: 0;
                li{
                  display: inline;
                  a{
                    text-decoration: none;
                    &:hover{}
                    color: red;
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}

```

```
}  
}
```

Após a compilação, com uma olhada no resultado, temos a ideia do monstro que acabamos de construir:

```
body .container .artigo .titulo h1 {  
  font-size: 20px; }  
body .container .artigo .titulo h2 {  
  font-size: 15px; }  
body .container .artigo .conteudo {  
  background-color: #CCC; }  
body .container .artigo .conteudo .secaoPrincipal {  
  font-size: 14px; }  
body .container .artigo .conteudo .secaoPrincipal .redes {  
  padding: 10px; }  
body .container .artigo .conteudo .secaoPrincipal .redes nav  
{  
  padding-top: 40px; }  
body .container .artigo .conteudo .secaoPrincipal .redes n  
av ul {  
  margin: 0; }  
body .container .artigo .conteudo .secaoPrincipal .redes  
nav ul li {  
  display: inline; }  
body .container .artigo .conteudo .secaoPrincipal .red  
es nav ul li a {  
  text-decoration: none;  
  color: red; }
```

O código exibido anteriormente deixa bem claro que, mesmo o Sass oferecendo aninhamento, não é uma boa ideia imitar o DOM na construção das regras de estilo. Esse tipo de abordagem oferece uma série de armadilhas. Seguem algumas:

- O desempenho na renderização cai drasticamente, uma vez que o CSS começa a leitura da direita para a esquerda.
- Muitos bytes serão adicionados ao arquivo CSS, já que o CSS nativo não aceita aninhamento e, no momento da compilação, teremos muitos seletores duplicados.
- Capacidade de manutenção é um problema, pois o CSS

está totalmente engessado ao HTML e qualquer alteração neste implica em alterações nas regras de estilo.

O recurso de aninhamento que o Sass oferece é extremamente útil. No entanto, é necessário ser cauteloso em seu uso. Embora não exista uma regra, não costumo usar mais de quatro níveis de aninhamento. Existem exceções, mas é importante que a maioria das regras de estilo sigam esta convenção.

3.7 UM PROJETO REAL

Continuando o projeto que foi iniciado no capítulo anterior, a ideia agora é usar os recursos do aninhamento para montar um menu lateral, como ilustrado a seguir.

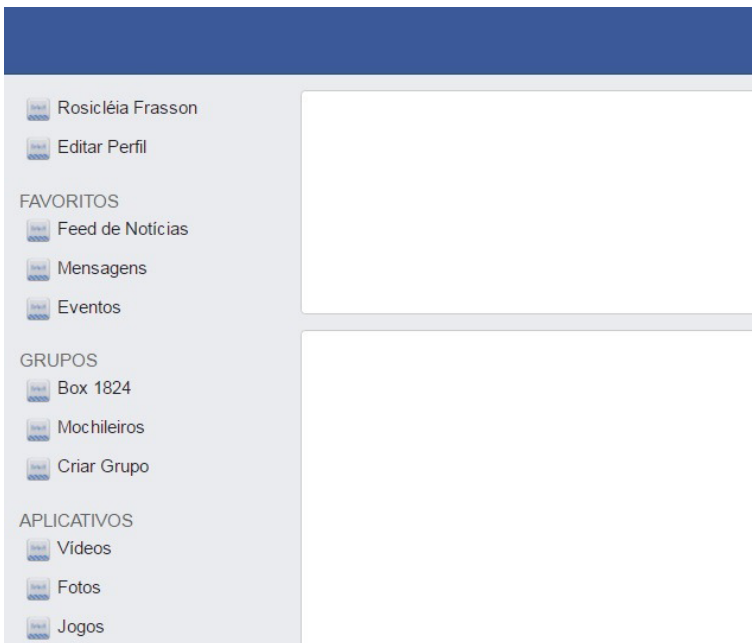


Figura 3.8: <https://goo.gl/3rKVdi>

A construção desse menu é muito similar ao construído no início do capítulo, e também usa uma lista para a sua construção. Veja um trecho da estrutura HTML para montá-lo.

```
<nav class="infoGerais">
  <ul>
    <li><a href="#">Rosicléia Frasson</a></li>
    <li><a href="#">Editar Perfil</a></li>
  </ul>
  <ul>
    <h4>Favoritos</h4>
    <li><a href="#">Feed de Notícias</a></li>
    <li><a href="#">Mensagens</a></li>
    <li><a href="#">Eventos</a></li>
  </ul>
  <ul>
    <h4>Grupos</h4>
    <li><a href="#">Box 1824</a></li>
    <li><a href="#">Mochileiros</a></li>
    <li><a href="#">Criar Grupo</a></li>
  </ul>
  ...
</nav>
```

Note que temos vários menus separados em categorias que estão definidas na tag `h4`.

Para a estilização, vamos usar o recurso de aninhamento, para que apenas os itens da lista do menu lateral sejam estilizados. Dessa forma, vamos encadear os elementos `ul`, `li` e `h4` dentro da classe `infoGerais`. Acompanhe o trecho de código a seguir que corresponde à estrutura de estilização do menu.

```
.infoGerais{
  ul{
    h4{
    }
    li{
      a{
        &:hover{
        }
      }
    }
  }
}
```

```
}
```

A construção de um menu com lista necessita que as formatações padrão dos elementos da lista sejam zeradas como o padding , a margin e a retirada das bolinhas que ficam em cada item da lista. O trecho de código exibido a seguir mostra como fazer a limpeza desses estilos padrões, e também estiliza o h4 que divide o menu em subseções.

```
.infoGerais{
  font-size: $tamanhoFonte;
  ul{
    list-style-type: none;
    padding: 0;
    margin: 0;
    h4{
      margin: 15px 0 0;
      font-weight: normal;
      text-transform: uppercase;
      color: #666;
    }
  }
}
```

Por fim, estilizaremos os itens de menu. Note que setamos a estilização para o link e adicionamos uma imagem antes de cada item de menu. No nosso exemplo, usamos a mesma imagem para todos os itens, mas poderíamos usar figuras distintas. O uso dessas estilizações resultará no menu exibido no início do tópico.

```
$corHoverPrincipal: #DDDFE2;
.infoGerais{
  font-size: $tamanhoFonte;
  ul{
    ...
  }
  li{
    a{
      width: 170px;
      text-decoration: none;
      display: inline-block;
      padding: 5px;
      border-radius: 3px;
      color: $corFontePrincipal;
```

```

        &:before {
            content: "";
            display: inline-block;
            vertical-align: middle;
            margin-right: 5px;
            width: 16px;
            height: 16px;
            background-image: url(image/defaultIcon.png);
        }
        &:hover{
            background-color: $corHoverPrincipal;
        }
    }
}

```

3.8 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Neste capítulo, foi visto como usar o recurso de aninhamento presente no Sass. Com ele, é possível aninhar regras de estilo, propriedades e grupos de seletores. Também foram abordados os tipos de combinação de seletores. Ao final do capítulo, foi feita uma reflexão sobre os perigos de usar demasiadamente o recurso de aninhamento.

O próximo capítulo fala sobre os mixins, um recurso extremamente poderoso para evitar a replicação de código e, consequentemente, facilitar o processo de manutenção em suas folhas de estilo.

REÚSE BLOCOS DE ESTILO COM MIXINS

As variáveis permitem que valores sejam reutilizados. No entanto, variáveis representam pedaços minúsculos em suas folhas de estilo, e precisamos de um mecanismo que permita reutilizar muito mais do que simples valores individuais.

A boa notícia é que o Sass também permite que blocos com regras de estilo sejam reutilizados. O Sass os chama de mixins.

O nome é bem sugestivo: um mixin mistura regras com outras regras. Em outras palavras, os mixins permitem a construção de blocos de código independentes, que você pode literalmente misturá-los para criar uma grande folha de estilo. Para quem conhece os conceitos de programação, pode pensar em um mixin como algo muito similar às funções das linguagens de programação.

O uso de mixins provê enormes vantagens, como a redução de código e a facilidade de manutenção, considerando que blocos semelhantes são escritos uma única vez e uma alteração feita em um bloco é propagada por toda a folha de estilo. Um mixin também pode ajudá-lo a deixar o código muito mais semântico, já que é possível separar as regras de estilo de acordo com a estrutura que elas representam e usá-las quando necessário.

4.1 DEFININDO UM MIXIN

Um mixin é declarado usando a diretiva `@mixin`, seguido pelo nome dado a ele. Os limites de um bloco de um mixin (ou seja, seu início e o fim) ficam entre chaves, e cada par propriedade e valor é finalizado com um ponto e vírgula, como mostrado a seguir.

```
@mixin nomeMixin{  
  propriedade: valor;  
  propriedade: valor;  
}
```

A regra de nomeação de mixins segue as mesmas regras de nomeação de variáveis, já mencionadas em capítulos anteriores. É muito importante que o nome de um mixin tenha uma relação estreita com o seu propósito. Isso facilita muito o processo de manutenção de suas folhas de estilo.

O uso de um mixin se justifica quando um bloco de regras de estilo precisa ser incluso em várias partes de um stylesheet. Ou seja, uma sequência de regras que se repetem em vários lugares é um bom candidato a virar um mixin.

É importante que você se atente ao agrupar regras de estilo em um mixin para que elas representem uma unidade lógica. Isto é, as regras agrupadas devem fazer sentido juntas.

Para exemplificar, imagine que a equipe de design definiu algumas configurações para as fontes que devem ser usadas em diversos blocos de sua aplicação web. Entre essas configurações, estão a família da fonte, peso e espaçamento entre as letras. É importante mencionar que nem todas as fontes da aplicação devem seguir este padrão, apenas algumas.

Com o uso de mixins, é possível agrupar esse conjunto de propriedades e valores para serem usados posteriormente em outras partes do seu stylesheet. Veja a seguir o mixin `fonteBase` que agrupa essas configurações.

```
@mixin fonteBase{
  letter-spacing: 0.2px;
  font-family: 'Calibri';
  font-weight: 400;
  line-height: 1.5;
}
```

4.2 INCLUIR UM MIXIN

Os mixins agrupam várias regras de estilo para serem aplicadas em partes distintas de um arquivo `.scss`. Sendo assim, para que os estilos sejam aplicados, é necessário que o mixin seja referenciado no arquivo.

Para usar um mixin, é necessário escrever a diretiva `@include`, seguido pelo nome do mixin, como mostrado a seguir.

```
seletor{
  @include nomeMixin;
  propriedade: valor;
}
```

Considerando que, em uma aplicação web, os botões e os parágrafos devem conter as regras de estilo definidas no mixin `fonteBase`, é necessário que ambos os seletores incluam este mixin. Veja:

```
p{
  @include fonteBase;
  color: #808080;
}

input[type="button"]{
  @include fonteBase;
  background: $corPrincipal;
  border-radius: 4px;
  ...
}
```

Note que, além da inclusão do mixin, outras regras foram adicionadas aos seletores. É importante perceber também que outros mecanismos do Sass podem trabalhar em conjunto com os

mixins, como o uso de variáveis e o aninhamento. No momento da compilação, todas as regras do mixin são adicionadas aos seletores. Veja a seguir:

```
p {
  letter-spacing: 0.2px;
  font-family: 'Calibri';
  font-weight: 400;
  line-height: 1.5;
  color: #808080; }

input[type="button"] {
  letter-spacing: 0.2px;
  font-family: 'Calibri';
  font-weight: 400;
  line-height: 1.5;
  background: #3E4FB2;
  border-radius: 4px;
  ... }
```

Com uma análise no CSS compilado, exibido anteriormente, é possível perceber que as regras de estilo que estavam dentro do mixin `fonteBase` foram inclusas nos dois seletores. Isso significa que os valores para as propriedades `letter-spacing`, `font-family`, `font-weight` e `line-height`, definidos dentro do mixin `fonteBase`, serão aplicados para os parágrafos e botões da página.

Vale ressaltar que, como o CSS nativo não entende os mixins, o arquivo compilado precisa duplicar o código. No entanto, como a manutenção acontece no arquivo `scss`, se for necessário alterar o valor de uma propriedade ou adicionar mais alguma a alteração, ocorre apenas no mixin.

O Sass também permite incluir mixins dentro de outros mixins. No trecho de código seguinte, está sendo definido um mixin com o nome de `conteudoBase`, que contém o mixin `fonteBase` definido anteriormente.

```
@mixin conteudoBase{
```

```
@include fonteBase;  
color: #808080;  
}
```

Um mixin pode ser declarado em qualquer parte do arquivo scss . No entanto, para utilizá-lo, é necessário que ele já tenha sido declarado.

Sass é inteligente o suficiente para compilar apenas os mixins que são usados em alguma parte da folha de estilo. Dessa forma, você pode criar bibliotecas de mixins e incluí-las em seus projetos, pois as regras existentes só serão compiladas se houver referência ao mixin.

Os mixins também podem ser declarados dentro de seletores, porém, estes só podem ser inclusos dentro dos seletores em que estão, ou nos seus descendentes, quando existirem regras aninhadas.

Outro ponto importante é que se as mesmas propriedades forem definidas após a inclusão de um mixin com outros valores, a página renderizada vai respeitar a última definição.

APLICANDO O PRÍNCÍPIO DRY NO CSS

DRY (*Don't Repeat Yourself*) quer dizer *não se repita*, em tradução livre. É um termo cunhado por Dave Thomas e Andy Hunt que virou uma filosofia seguida pelos programadores.

A ideia por trás deste conceito consiste em ter uma representação única de um trecho de código. Isto é, se você já programou um trecho de código que resolve uma dada situação e, em outro momento, precisa do mesmo trecho de código para resolver uma outra situação, você não pode dar Ctrl+C e Ctrl+V no código. Você precisa reutilizar o código já escrito.

Este conceito visa reduzir a duplicação de código e, consequentemente, os problemas de manutenção resultantes. Código duplicado significa retrabalho no momento da manutenção, além de gerar contradições lógicas.

Por muito tempo, este princípio não pode ser aplicado na construção de folhas de estilo, já que o CSS nativo não suporta nenhum tipo de função, procedimento ou mecanismo similar. Com a chegada dos pré-processadores, a realidade mudou e, atualmente, é possível aplicar algumas boas práticas de programação na construção de stylesheets. O DRY é uma delas e pode ser aplicado com Sass usando mixins.

4.3 USO DE PARÂMETROS

Muitas vezes em nossas folhas de estilo, temos blocos de regras que apresentam grande similaridade, com apenas algumas particularidades nos valores das propriedades. Nesses casos, é

possível incluir argumentos em um mixin para permitir que um bloco seja customizado no momento da compilação.

Os argumentos são variáveis definidas juntamente com a definição do mixin, seguidos do seu nome e delimitados pelos parênteses. Quando existir mais de um argumento, estes devem estar separados por vírgula. Veja:

```
@mixin nomeMixin ($argumento1, $argumento2){  
  propriedade: $argumento1;  
  propriedade: valor;  
  propriedade: $argumento2;  
}
```

Imagine que os botões precisam ser mais expressivos na página, necessitam de um peso de fonte maior e, consequentemente, aumentar o espaçamento das letras para que continuem legíveis. Para aproveitar o mixin, é possível adicionar argumentos para que ele consiga estilizar os parágrafos e os botões.

No trecho de código a seguir, no mixin `fonteBase` foram definidos dois argumentos que são utilizados pelas propriedades definidas dentro do mixin:

```
@mixin fonteBase ($pesoFonte, $espacamentoLetras){  
  letter-spacing: $espacamentoLetras;  
  font-family: 'Calibri';  
  font-weight: $pesoFonte;  
  line-height: 1.5;  
}
```

Como o mixin agora possui argumentos, no momento da sua utilização é necessário definir os valores que eles devem usar. É importante que a ordem dos valores passados no momento da utilização do mixin seja a mesma em que os argumentos foram declarados.

No trecho de código exibido adiante, é possível perceber que, para o seletor `p`, o mixin `fonteBase` terá o valor 400 para o peso

da fonte e 0.2px de espaçamento entre as letras. Já nos botões o peso da fonte, será 600 e o espaçamento entre as letras de 0.6px.

```
p{
  @include fonteBase(400, 0.2px);
  color: #808080;
}

input[type="button"]{
  @include fonteBase (600, 0.6px);
  background: $corPrincipal;
}
```

O processo de compilação deve gerar blocos customizados de regras CSS com os valores passados. O CSS seguinte exibe o código compilado.

```
p {
  letter-spacing: 0.2px;
  font-family: 'Calibri';
  font-weight: 400;
  line-height: 1.5;
  color: #808080; }

input[type="button"] {
  letter-spacing: 0.6px;
  font-family: 'Calibri';
  font-weight: 600;
  line-height: 1.5;
  background: #3E4FB2;}
```

A figura adiante ilustra o resultado do código renderizado no navegador, e é possível notar a diferença entre o peso e a fonte do parágrafo e do botão.

O processador do Sass é escrito em
conhecimento na linguagem para
diretório de desenvolvimento está
Podemos contribuir para a sua mel
recursos e codificação.

Saiba mais

Figura 4.1: <https://goo.gl/FXyrbN>

Também é possível ter um valor default associado a um argumento do mixin. Neste caso, se nenhum valor for definido, o valor default é aplicado. No trecho de código seguinte, o argumento `$espacamentoLetras` possui o valor `0.2px` definido.

```
@mixin fonteBase ($pesoFonte, $espacamentoLetras: 0.2px){  
  letter-spacing: $espacamentoLetras;  
  font-family: 'Calibri';  
  font-weight: $pesoFonte;  
  line-height: 1.5;  
}
```

Note a seguir que o uso do mixin nos parágrafos não possui valor para `$espacamentoLetras`. Já no uso do mixin para os botões, o espaçamento foi definido em `0.6px`.

```
p{  
  @include fonteBase(400);  
  color: #808080;  
}  
  
input[type="button"]{  
  @include fonteBase (600, 0.6px);  
  background: $corPrincipal;  
}
```

Na compilação, o `letter-spacing` dos parágrafos é

substituído pelo valor default (0.2px) e, nos botões, é o valor definido no momento da inclusão do mixin (0.6px). O trecho compilado está exibido a seguir.

```
p {  
  letter-spacing: 0.2px;  
  font-family: 'Calibri';  
  font-weight: 400;  
  line-height: 1.5;  
  color: #808080; }  
  
input[type="button"] {  
  letter-spacing: 0.6px;  
  font-family: 'Calibri';  
  font-weight: 600;  
  line-height: 1.5;  
  background: #3E4FB2;}
```

Vale ressaltar que a composição de mixins com variáveis aumenta expressivamente a capacidade de reúso com o Sass.

4.4 MIXIN VENDOR PREFIX

O CSS3 trouxe muitas novidades, dentre elas `gradient`, `box-shadow` e `transformation`. No entanto, elas não se encontram completamente definidas e implementadas nos navegadores.

Durante a fase de transição, é possível utilizar versões experimentais dessas funcionalidades. Para que isso seja possível, os navegadores costumam adicioná-las com prefixos específicos para diferenciar da versão final.

O grande problema é que os navegadores ocasionalmente usam sintaxes diferentes para as funcionalidades experimentais. Isso obriga os desenvolvedores a escrever várias linhas de código para definir uma mesma propriedade.

Os mixins podem ser usados para agrupar os prefixos. E quando o código é compilado para o CSS, todos os prefixos são

automaticamente gerados. Veja o caso do `border-radius` :

```
@mixin bordaArredondada ($tamanho) {  
  -moz-border-radius: $tamanho;  
  -webkit-border-radius: $tamanho;  
  border-radius: $tamanho;  
}
```

O uso deste tipo de mixin, além de reduzir a quantidade de código duplicado, facilita bastante a adição do suporte a uma versão diferente de um navegador, visto que a adição do prefixo ocorre apenas no mixin. No momento em que não forem mais necessários os prefixos, a remoção também é facilitada, considerando que a alteração será feita em apenas um lugar.

4.5 REGRAS CSS NOS MIXINS

Mixins podem conter mais do que propriedades. É possível que regras sejam adicionadas, com seletores e propriedades.

Imagine que você precise estilizar vários tipos de cards, porém o título deles é bem similar. É possível montar um mixin para o título que contém mais de um seletor, e usá-lo em todos os tipos de cards. Veja no trecho de código a seguir que, dentro do mixin `tituloCard`, existem regras para os seletores `h1` e `.card-titulo`.

```
@mixin tituloCard($corFundoTitulo){  
  .card-titulo{  
    padding: 5px;  
    background: $corFundoTitulo;  
    text-align: center;  
    h1{  
      color: white;  
      font-size: 40px;  
    }  
  }  
}
```

É importante ressaltar que os mixins podem conter variáveis

declaradas dentro deles, e também permitem todas as regras de aninhamento já vistas. Diferente do que aconteceu em capítulos anteriores, neste, todos os mixins apresentados foram usados na confecção de cards.

O código completo pode ser consultado no GitHub do livro. A figura a seguir representa a renderização de dois cards que usam os mixins vistos neste capítulo.



Figura 4.2: <https://goo.gl/guxMx5>

4.6 FAZENDO MUITO MAIS COM MIXINS

Os mixins apresentados neste capítulo podem ser considerados mixins simples. É importante que você saiba que o poder dos mixins vai muito além. Os próximos parágrafos citam mixins muito mais poderosos e úteis que podem ser usados na construção de suas folhas de estilo.

Centralizando um elemento

Muitas vezes, em nossas páginas web, precisamos que um elemento fique centralizado em relação ao elemento pai. Para conseguir este efeito, precisamos de algumas propriedades CSS. Como usamos rotineiramente este efeito, é muito interessante que isto vire um mixin. O trecho de código seguinte exhibe-o:

```
@mixin centralizado {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}
```

Escondendo o texto excedente

Em aplicações web, quando o texto é alimentado pela aplicação, não é possível prever o seu tamanho. E se o texto for grande demais, o layout acaba quebrando. Uma forma de resolver o problema é trocar o texto excedente por `ellipsis`. Para isso, é necessário usar um conjunto de propriedades: `white-space`, `overflow` e `text-overflow`.

Na grande maioria das aplicações, esta técnica é usada em muitos elementos. Portanto, é interessante transformar esse conjunto de valores em um mixin.

```
@mixin ellipsis{
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;
}
```

Usando fontes customizadas

A propriedade `@font-face` permite a utilização de famílias de fontes fora do padrão do sistema operacional. Para que elas sejam aplicadas, é necessário incluir o endereço onde as fontes estão alocadas.

Normalmente, você utiliza mais de uma família de fonte em sua aplicação. Se for necessário suportar vários browsers, é necessário apontar os arquivos de cada formato suportado por cada navegador para cada família de fonte. Isso significa muitas linhas de código praticamente iguais sendo replicadas muitas vezes.

Para contornar este problema, um mixin pode ajudá-lo. Veja:

```
@mixin fonteCustomizada($font-family, $font-url, $font-name, $weight) {
  @font-face {
    font: {
      family: $font-family;
      style: normal;
      weight: $weight;
    }

    src: url($font-url + '/' + $font-name + '.eot') format('eot'),
         url($font-url + '/' + $font-name + '.eot?#iefix') format('embedded-opentype'),
         url($font-url + '/' + $font-name + '.woff') format('woff'),
         url($font-url + '/' + $font-name + '.ttf') format('true-type');
  }
}
```

Note que o mixin se encarrega de montar, para cada família de fonte, vários formatos usados em diferentes navegadores. No momento da utilização, é necessário apenas completar os parâmetros que o mixin possui.

O código a seguir monta os valores da propriedade `@font-face` para duas famílias de fontes. Note que, em ambos os casos, no momento da utilização foi definido o nome da família, o caminho do arquivo que representa a fonte, o nome do arquivo e o peso da fonte. Essas definições correspondem aos parâmetros exigidos na definição do mixin.

```
@include fonteCustomizada('scarface-webfont', 'fonts/scarface/', 'scarface', 400);
@include fonteCustomizada('futura-webfont', 'fonts/futura/', 'futura', 200);
```

4.7 UM PROJETO REAL

Neste capítulo, vamos incrementar um pouco mais o nosso

projeto com o uso de alguns mixins. Em páginas nas quais as informações são provenientes da interação do usuário com a aplicação, não temos controle do conteúdo a ser exibido. Sendo assim, podemos separar um espaço para um determinado conteúdo e ele não ser suficiente quebrando o nosso layout. Veja a figura a seguir:



Figura 4.3: <https://goo.gl/dSCgDE>

Você pode observar nessa figura que temos dois itens de menu em que o conteúdo extrapolou os limites determinados. Veja que, nos dois casos, não temos controle da informação. Ela será diferente para cada usuário.

Uma das formas de contornar este problema é limitar o tamanho do texto, trocando o excedente por ellipsis, como já foi sugerido neste capítulo. Vamos então incluir o mixin `ellipsis` na estilização do link, como mostra o código seguinte. Lembre-se de incluir o mixin na sua folha de estilo, preferencialmente no início dela.

```
@mixin ellipsis{
  overflow: hidden;
  white-space: nowrap;
  text-overflow: ellipsis;
}
```

```

a{
  width: 170px;
  text-decoration: none;
  display: inline-block;
  padding: 5px;
  border-radius: 3px;
  color: $corFontePrincipal;
  @include ellipsis;
}

```

Se renderizarmos a página no browser, o resultado deve ser:

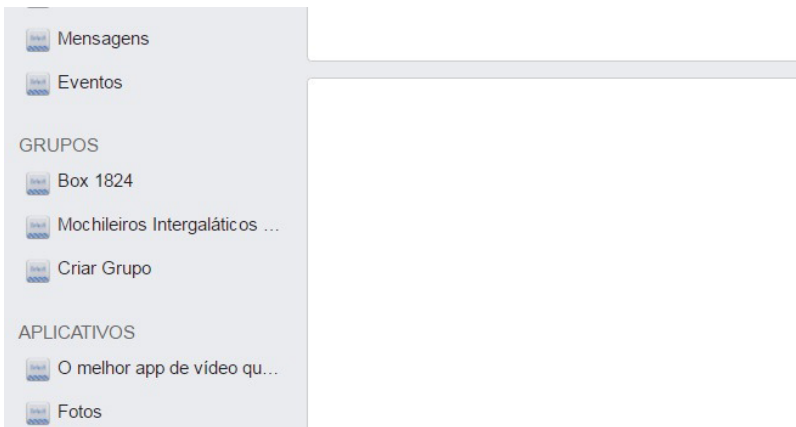


Figura 4.4: <https://goo.gl/ofMEfF>

Para melhorar a manutenção no nosso projeto, vamos construir um mixin para estilizar as fontes. As propriedades que normalmente são estilizadas são a cor, o tamanho, o peso e o controle de entrelinhas. Acompanhe a criação do mixin no trecho de código seguinte. Note que algumas propriedades possuem um valor default, mas que pode ser sobrescrito.

```

@mixin estiloFonte($cor, $tamanho:12px, $peso:normal, $entrelinhas:
22px){
  color: $cor;
  font-size: $tamanho;
  font-weight: $peso;
  line-height: $entrelinhas;
}

```

Você pode usar este mixin para estilizar as fontes presentes nas tags `h4` e `a`, como exibido a seguir. Nos próximos capítulos, usaremos este mixin para estilizar as demais fontes da página.

```
h4{
    margin: 15px 0 0;
    @include estiloFonte($cor: #666);
    text-transform: uppercase;
}

a{
    width: 170px;
    text-decoration: none;
    display: inline-block;
    padding: 5px;
    border-radius: 3px;
    @include estiloFonte($cor: $corFontePrincipal, $entrelinhas: 1
6px);
}
```

4.8 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Este capítulo mostrou como usar os mixins para deixar suas folhas de estilo mais flexíveis e escalares. Nele foram vistos como criar e usar mixins simples e com o uso de parâmetros. Também foi demonstrado como usar os mixins para resolver o problema do `vendor prefix`. No final, alguns mixins poderosos foram apresentados como exemplos de utilização da técnica.

No próximo capítulo, será abordada a técnica de herança, que consiste em mais uma forma de evitar a replicação de código.

EVITE PROPRIEDADES DUPLICADAS COM HERANÇA

As aplicações web são compostas por muitos elementos que mantêm um padrão visual e estético. Isso significa que muitos elementos possuem características visuais ou estruturais semelhantes.

Os mixins consistem em uma forma de agrupar essas características em blocos de código para uma posterior reutilização. No entanto, é bom atentar-se ao fato da repetição de código no arquivo gerado.

Suponha que você possua um mixin que é usado quinze vezes na sua folha de estilo. Após a compilação, é possível observar que as regras que existiam dentro do mixin foram escritas quinze vezes. Isso pode significar um grande problema se você necessita reduzir o tamanho do arquivo para diminuir a velocidade de carregamento.

O Sass possui um mecanismo chamado herança, que permite a reutilização de blocos de código sem que estes sejam repetidos várias vezes no momento da compilação. Ou seja, o compilador do Sass não copia as regras, ele apenas as separa.

5.1 USANDO A HERANÇA

A herança instrui um seletor a usar todos os estilos de um outro seletor sem duplicar as propriedades CSS. Isso é muito útil quando elementos compartilham características genéricas com algumas particularidades. No Sass, para conseguir esse comportamento, é usada a diretiva `@extend`.

A diretiva `@extend` é utilizada para estender outros estilos. Ela permite que um seletor herde propriedades e valores declarados em outro. Para usar este recurso, basta colocar a diretiva em conjunto com o seletor que deve ser herdado. Veja a seguir:

```
.qualquerSeletor{
  @extend seletorHerdado;
  propriedade: valor;
}
```

Para exemplificar, suponha que uma aplicação web possua algumas variações para os botões. Além do botão tradicional, existem o botão com a borda arredondada e o com a altura um pouco maior. No caso citado, os botões citados são muito semelhantes, apenas algumas características os diferenciam. Sendo assim, é possível fazer o uso da herança.

No código exibido a seguir, as regras que são comuns foram colocadas dentro do seletor `.botao` e representam o botão tradicional. Os botões arredondados são estilizados pelo seletor `.botaoRedondo`, e os botões maiores pelo seletor `.botaoLargo`.

Note que, em ambos os seletores, foi usada a diretiva `@extend` seguida pelo seletor `.botao`. Isso significa que todas as regras presentes no botão tradicional também devem ser aplicadas aos botões arredondado e largo.

```
.botao {
  border:none;
  padding: 8px;
  min-width: 80px;
  text-align: center;
  cursor: pointer;
}
```

```

    font-size: 16px;
    color: #FFFFFF;
    background-color: #0096ED;
    box-shadow: 0 1px #666666, 0 -1px #53BFFF;
    &:hover{
        background-color: #53BFFF;
    }
}

.botaoRedondo{
    @extend .botao;
    border-radius: 4px;
}

.botaoLargo{
    @extend .botao;
    height: 40px;
}

```

Vale ressaltar que todos os outros mecanismos presentes no Sass podem ser utilizados em conjunto com a herança. Note no exemplo anterior que foi usado o recurso de aninhamento. Poderíamos usar variáveis e mixins também.

O CSS adiante corresponde ao código compilado. É importante perceber que as regras comuns não foram duplicadas. Para conseguir o efeito desejado, o Sass isola as regras genéricas e usa um conjunto de seletores para aplicá-las.

```

.botao, .botaoRedondo, .botaoLargo {
    border: none;
    padding: 8px;
    min-width: 80px;
    text-align: center;
    cursor: pointer;
    font-size: 16px;
    color: #FFFFFF;
    background-color: #0096ED;
    box-shadow: 0 1px #666666, 0 -1px #53BFFF; }
.botao:hover, .botaoRedondo:hover, .botaoLargo:hover {
    background-color: #53BFFF; }

.botaoRedondo {
    border-radius: 4px; }

```

```
.botaoLargo {
  height: 40px; }
```

É importante ressaltar que os três seletores podem ser usados como classes. O efeito renderizado no browser pode ser visto na figura a seguir.



Figura 5.1: <https://goo.gl/eQtx39>

5.2 PLACEHOLDER

Existem alguns casos em que declaramos um seletor com o único intuito de estender para outros. Nas situações em que definimos regras apenas para estendê-las, é possível usar o seletor placeholder .

Um placeholder não pode ser usado como uma classe CSS. Eles são uma alternativa segura para você construir estilos comuns e garantir que eles somente serão compilados se forem referenciados.

O seletor placeholder é iniciado com o símbolo % (porcentagem). No momento de usá-lo, também é necessário acrescentar o símbolo juntamente com @extend .

```
%placeholderDeclarado{
  propriedade: valor;
  propriedade: valor;
}

.qualquerSeletor{
  @extend %placeholderDeclarado;
  propriedade: valor;
}
```

Um bom exemplo para o uso de placeholder são as notificações de uma aplicação web. Geralmente, alguns tipos de notificações ou alertas podem ser exibidos para o usuário na realização de uma

ação, como alerta de sucesso, de erro, um aviso ou um alerta de informação. Em sua grande maioria, essas notificações são bem similares.

Na ampla maioria dos sistemas web, os tipos de notificações costumam ter a mesma estrutura, diferindo geralmente apenas nas cores e ícones usados. No exemplo de conjunto de notificações que vem a seguir, a diferença fica apenas para o padrão de cores, o restante das estilizações é o mesmo.

Isso indica que é possível utilizar um placeholder para fazer todo o estilo de todos os tipos de notificações, exceto o padrão de cores, como mostrado no trecho de código adiante.

```
%alerta{
  width: 80%;
  margin: 20px auto;
  padding: 20px;
  box-shadow: 0 0 8px rgba(0,0,0,.3);
  border-left-width: 8px;
  border-left-style: solid;
  border-radius: 5px;
}
```

É importante ressaltar que nenhum dos tipos de notificação terá apenas estas regras, portanto, não é necessário que exista um seletor `.alerta`. Todos os tipos de notificação herdam as características do placeholder `alerta` e incluem suas próprias características, como pode ser visto no trecho a seguir.

```
.alertaErro{
  @extend %alerta;
  background-color: tomato;
  color: #FFFFFF;
  border-left-color: #DC3D21
}

.alertaWarning{
  @extend %alerta;
  background-color: #EAAF51;
  color: #6B6D31;
  border-left-color: #DF8B00;
```

```

}

.alertaSucesso{
    @extend %alerta;
    background-color: #61B832;
    color: #296829;
    border-left-color: #55A12C;
}

.alertaInfo{
    @extend %alerta;
    background-color: #4EA5CD;
    color: #BEECF6;
    border-left-color: #3B8EB5;
}

```

Vale ressaltar que as propriedades não serão necessariamente iguais como no exemplo apresentado. Foi uma mera coincidência.

No momento da compilação, um conjunto de seletores aplica as regras genéricas, e as regras específicas são aplicadas isoladamente para cada seletor. Note no trecho de CSS compilado que não existe o seletor `.alerta`. Essa é a diferença entre a herança tradicional e o uso de placeholder.

```

.alertaErro, .alertaWarning, .alertaSucesso, .alertaInfo {
    width: 80%;
    margin: 20px auto;
    padding: 20px;
    box-shadow: 0 0 8px rgba(0, 0, 0, 0.3);
    border-left-width: 8px;
    border-left-style: solid;
    border-radius: 5px; }

.alertaErro {
    background-color: tomato;
    color: #FFFFFF;
    border-left-color: #DC3D21; }

.alertaWarning {
    background-color: #EAAF51;
    color: #6B6D31;
    border-left-color: #DF8B00; }

.alertaSucesso {
    background-color: #61B832;

```

```
color: #296829;
border-left-color: #55A12C; }

.alertaInfo {
background-color: #4EA5CD;
color: #BEECFD;
border-left-color: #3B8EB5; }
```

O resultado do código de exemplo são as caixas de notificação, ilustradas na figura seguinte.



Figura 5.2: <https://goo.gl/35FHp9>

5.3 A RELAÇÃO ÍNTIMA ENTRE HERANÇA E OOCSS

Quando estilizamos um elemento, aplicamos características visuais e estruturais. Como características visuais, considere cores, fontes, gradientes, bordas, entre outros. Já as características estruturais podem ser representadas pelo tamanho, distância e posição. O fato é que tanto as características visuais quanto as estruturais são repetidas diversas vezes em muitos elementos da aplicação.

Nicole Sullivan introduziu em 2008 o termo OOCSS (Object Oriented CSS), que basicamente sugere que as características visuais

fiquem separadas das estruturais, para que diferentes elementos possam reutilizá-las.

Se considerarmos uma aplicação web, muitos elementos compartilham a mesma estrutura ou tem características visuais semelhantes. A título de exemplo, vamos considerar dois componentes que estão presentes na grande maioria das páginas web: caixas de mensagens e botões.

Podemos ter diversos tipos de caixas de mensagens, porém suas estruturas devem ser bem similares. O mesmo ocorre com botões. Também podemos ter características visuais idênticas em um botão e uma caixa de mensagem. Observe a figura a seguir.

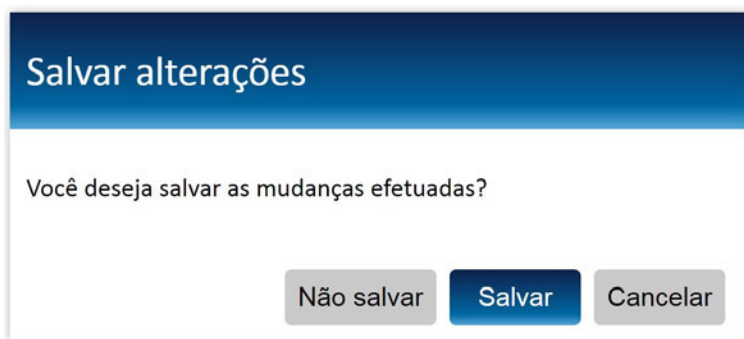


Figura 5.3: <https://goo.gl/TovZ9P>

Nela é possível perceber que o cabeçalho da caixa de mensagem possui características visuais semelhantes às do botão salvar. Também é possível perceber que os três botões visíveis na tela possuem a mesma característica estrutural. Nestes casos, podemos fazer uso dos conceitos de OOCSS para reaproveitamento de código.

O código exibido na sequência representa uma parte da estilização da caixa de diálogo mostrada anteriormente. É possível identificar a estilização para três seletores.

O seletor `.botao` contém as características estruturais para os três botões presentes na imagem. O seletor `.fundoPadrao` possui a característica visual para os botões que estão na cor cinza, e poderia ser usado em outros componentes que precisassem da mesma característica. Já o seletor `.fundoAzul` possui as características visuais que são usadas no botão salvar e no cabeçalho da caixa de diálogo.

```
.botao{
  border:none;
  padding: 8px;
  min-width: 80px;
  text-align: center;
  cursor: pointer;
  font-size: 16px;
  margin: 4px;
  border-radius: 4px;
}

.fundoPadrao{
  background-color: #CACACA;
}

.fundoAzul{
  background: linear-gradient(to bottom, #0f204b 0%,#0066a1 80%,
#53a7d9 100%);
  color: #FFFFFF;
}
```

É importante perceber que regras declaradas em seletores diferentes precisam ser aplicadas no mesmo elemento. Note no trecho de HTML seguinte que a definição de classes CSS é um pouco diferente. Na `div` que representa o cabeçalho da caixa de mensagem, temos duas classes CSS: `cabecalho` e `fundoAzul`. O mesmo acontece com os botões que possuem as classes `botao` e `fundoAzul` ou `fundoPadrao`.

```
<div class="dialog">
  <div class="cabecalho fundoAzul">
    <span>Salvar alterações</span>
  </div>
  <div class="conteudo">
```

```

        <p>Você deseja salvar as mudanças efetuadas?</p>
    </div>
    <div class="rodape">
        <button class="botao fundoPadrao">Não salvar</button>
        <button class="botao fundoAzul">Salvar</button>
        <button class="botao fundoPadrao">Cancelar</button>
    </div>
</div>

```

Vale ressaltar que poderíamos ter quantas classes CSS fossem necessárias. Poderíamos ter, por exemplo, uma classe `iconeSalvar` que poderia ser aplicada no botão e em qualquer outro elemento. Poderíamos também ter uma classe que representasse os estilos de um botão diferenciado como `botaoGigante`.

A técnica de OOCSS é aplicada em grandes frameworks CSS do mercado, como o Bootstrap e o Foundation. No entanto, na minha opinião, o uso da técnica como foi concebida originalmente impõe uma interferência muito grande na marcação HTML, já que é necessário adicionar várias classes CSS no mesmo elemento. A boa notícia é que, com o auxílio do Sass, é possível melhorar a técnica usando-a sem interferência extrema no HTML. Veja:

```

%botao{
    border:none;
    padding: 8px;
    min-width: 80px;
    text-align: center;
    cursor: pointer;
    font-size: 16px;
    margin: 4px;
    border-radius: 4px;
}

%fundoPadrao{
    background-color: #CACACA;
}

%fundoAzul{
    background: linear-gradient(to bottom, #0f204b 0%,#0066a1 80%,
#53a7d9 100%);
    color: #FFFFFF;
}

```

```
}
```

O primeiro passo é transformar os seletores que representavam características visuais e estruturais em placeholders. Esses placeholders são usados nos seletores que necessitam das respectivas estilizações. Observe no trecho de código seguinte:

```
.cabecalhoAzul{
  padding: 20px 10px;
  font-size: 25px;
  @extend %fundoAzul;
}

.botao{
  &Padrao{
    @extend %botao;
    @extend %fundoPadrao;
  }
  &Azul{
    @extend %botao;
    @extend %fundoAzul;
  }
}
```

Note que o cabeçalho estendeu o placeholder `fundoAzul`. Na estilização dos botões, os placeholders foram usados em conjunto com o recurso de aninhamento para conseguir o efeito desejado.

O CSS resultante pode ser visto a seguir. Note que o Sass construiu vários seletores: `.botaoAzul`, `.botaoPadrao` e `cabecalhoAzul`. Com estes, é possível estilizar os elementos sem uma intervenção extrema na marcação HTML.

```
.botaoPadrao, .botaoAzul {
  border: none;
  padding: 8px;
  min-width: 80px;
  text-align: center;
  cursor: pointer;
  font-size: 16px;
  margin: 4px;
  border-radius: 4px; }

.botaoPadrao {
```

```
background-color: #CACACA; }

.cabecalhoAzul, .botaoAzul {
  background: linear-gradient(to bottom, #0f204b 0%, #0066a1 80%,
#53a7d9 100%);
  color: #FFFFFF; }

.cabecalhoAzul {
  padding: 20px 10px;
  font-size: 25px; }
```

O trecho do HTML pode ser conferido a seguir. É possível observar que não foi necessário usar mais de uma classe CSS para cada elemento.

```
<div class="dialog">
  <div class="cabecalhoAzul">
    <span>Salvar alterações</span>
  </div>
  <div class="conteudo">
    <p>Você deseja salvar as mudanças efetuadas?</p>
  </div>
  <div class="rodape">
    <button class="botaoPadrao">Não salvar</button>
    <button class="botaoAzul">Salvar</button>
    <button class="botaoPadrao">Cancelar</button>
  </div>
</div>
```

Independente de usar a OOCSS original ou a OOCSS com Sass, é importante aplicar a técnica com cautela. Apenas características que são repetidas diversas vezes e constituem um padrão da aplicação devem ser separadas dessa forma.

5.4 HERANÇA VERSUS MIXIN

No momento da construção de suas folhas de estilo, uma dúvida muito recorrente é se deve ser usado um mixin para o bloco de código ou a herança. É importante ressaltar que não existem regras fechadas para ajudá-lo na decisão. No entanto, valem algumas considerações:

- O uso de placeholders ou de herança com seletor não permite o uso de parâmetros. Isso significa que, se você possui trechos de código com propriedades iguais, mas com valores diferentes, a única alternativa para reaproveitamento de código é com o uso de mixins.
- Se você precisa diminuir o tamanho do arquivo CSS, prefira usar a herança em vez de mixins.
- Quando for usar herança, prefira usá-la com placeholder. A manutenção é mais simples e você evita ter regras sobrescritas.
- Tanto com o uso de mixins ou de herança quanto propriedades ou regras são sobrescritas. O browser vai renderizar a última propriedade ou regra que aparecer. Isso significa que, caso seu mixin atribua 12px para o tamanho da fonte e após a sua inclusão uma regra de tamanho de fonte for adicionada com um valor de 16px, a fonte do elemento será renderizada com 16px.
- Tanto a herança quanto os mixins são recursos extremamente poderosos, e o uso de ambos facilitará a escrita do seu CSS. Um pouco de experiência e análise no código gerado lhe ajudarão a decidir em quais momentos cada um é recomendado.

5.5 UM PROJETO REAL

Com as funcionalidades apresentadas neste capítulo, é possível construir os modelos de botões que necessitamos para o nosso projeto. A figura seguinte possui dois tipos de botões que construiremos a seguir. O restante da estilização do bloco você pode acompanhar no repositório de código do livro.



Figura 5.4: <https://goo.gl/wYyQzO>

Inicialmente, vamos construir uma base para todos os botões. Usaremos o placeholder para isso. Veja:

```
%botao{
  border-radius: 2px;
  height: 24px;
  margin-right: 10px;
}
```

O botão cinza foi chamado de `botaoPadrao` e estende as estilizações feitas no `%botao`. Em seguida, adicionamos as estilizações que são exclusivas deste botão. Acompanhe a seguir:

```
.botaoPadrao{
  @extend %botao;
  background: $botaoPadrao;
  border: 1px solid $botaoPadraoBorda;
  @include estiloFonte($cor:$botaoPadraoFonte, $peso:bold);
  display: inline-block;
}
```

Este botão também possui um ícone, precisamos adicioná-lo. Note que o pseudoelemento `:before` abre espaço para o ícone, mas não temos referência para ele. Essa referência é adicionada em uma classe CSS separada. Neste caso, é chamada de `iconeMundo`. Trabalhando dessa forma, podemos ter o mesmo botão cinza com ícones diferentes.

```
.botaoPadrao{
  ...
  &:before{
    content: "";
```

```

        position: absolute;
        margin-right: 5px;
        width: 12px;
        height: 12px;
        margin: 4px;
    }
    span{
        margin-left: 24px;
    }
}
.iconeMundo:before{
    background-image: url(image/mundo.png);
}

```

Por fim, veja o estilo para o comportamento de hover do botão.

```

.botaoPadrao{
    ...

    &:hover{
        background-color: $botaoPadraoHover;
    }
}

```

O botão azul, por não possuir ícone, é muito mais simples. Apenas estendemos do placeholder `botao` e adicionamos as estilizações de fundo, borda, fonte e hover. Veja o código seguinte:

```

.botaoDestaque{
    @extend %botao;
    background: $botaoDestaque;
    border: 1px solid $botaoDestaqueBorda;
    @include estiloFonte($cor:$botaoDestaqueFonte, $peso:bold);

    &:hover{
        background-color: $botaoDestaqueHover;
    }
}

```

Para que o botão renderize na tela, como na imagem mostrada no início do tópico, é necessário que o HTML seja similar ao trecho seguinte:

```

<button class="botaoPadrao iconeMundo"><span>Público</span></button>
<button class="botaoDestaque"><span>Publicar</span></button>

```

5.6 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Com o uso de herança e mixins, a reutilização de código se torna real, os problemas de manutenção são reduzidos e a velocidade de carregamento da página melhora. É importante se atentar para o uso correto dos recursos apresentados. Enquanto você estiver na fase de aprendizado, é muito recomendado que você efetue auditorias no código gerado para certificar-se de ter tomado boas decisões.

Vale ressaltar que, embora a herança seja um recurso extremamente poderoso, se for aplicado em excesso pode se tornar extremamente nocivo.

No próximo capítulo, será apresentado o recurso de particionamento de arquivos. Recurso este extremamente importante para deixar seu CSS modular.

DIVIDIR PARA CONQUISTAR

Quem conhece CSS há algum tempo já deve ter visto o particionamento de um grande arquivo CSS em arquivos pequenos para facilitar a manutenção. Em páginas web simples, por exemplo, normalmente existe um arquivo para o cabeçalho da página, um outro para o rodapé e ainda um arquivo para menu. E para que a página web seja estilizada por completo, existe um arquivo principal que importa todos esses arquivos, como mostrado a seguir:

```
@import url("cabecalho.css");  
@import url("rodape.css");  
@import url("menu.css");
```

Essa estratégia permite que o código fique particionado em vários arquivos. Entretanto, existe um grande problema de performance nessa abordagem.

Isso acontece porque cada arquivo usado na página, seja ele HTML, CSS, JavaScript ou um arquivo de imagem, é carregado pelo navegador através de uma requisição ao servidor. E quanto maior o número de requisições, mais demorado é o carregamento da página. Sendo assim, é possível usar a diretiva `@import` do CSS. Entretanto, essa não é considerada uma boa prática, devido aos problemas de desempenho no carregamento da página.

Atualmente, existem formas de trabalhar com arquivos particionados no ambiente de desenvolvimento e, em ambiente de

produção, fazer a junção deles para reduzir a quantidade de requisições. O Sass oferece um meio para se trabalhar dessa forma.

No Sass, você pode separar seu código `scss` em muitos arquivos distintos. No momento da compilação, apenas um arquivo é criado com a junção de todos os arquivos. Dessa forma, temos um CSS modular no ambiente de desenvolvimento e apenas um único arquivo no ambiente de produção, reduzindo assim o número de requests feitos ao servidor.

6.1 CRIANDO UM ARQUIVO PARTICIONADO

O primeiro passo para se trabalhar com arquivos particionados é obviamente ter um arquivo com a extensão `.scss`. Nele podem ser declarados variáveis, seletores com regras, mixins, placeholders, e você pode usar o recurso de aninhamento. Enfim, todos os recursos presentes no CSS e no Sass podem ser usados em arquivos particionados.

É importante ter um padrão para representar os arquivos que representam partes e o arquivo principal que contém a importação das partes. Se você não deseja que um arquivo Sass se transforme em um arquivo CSS no momento da compilação (ou seja, se o arquivo representar apenas uma parte de seu stylesheet), é necessário que o nome do mesmo comece com o caractere `_` (underscore).

Para exemplificar, vamos transformar a estilização dos botões e das notificações apresentadas no capítulo anterior em partes. Portanto, devemos ter dois arquivos com os nomes `_botoes.scss` e `_notificacoes.scss`, como mostrado a seguir.

```
|-- _botoes.scss  
|-- _notificacoes.scss
```

É importante perceber que, para melhorar a produtividade e

facilitar a manutenção, é necessário que você estipule regras para a separação. Isso vai depender muito do contexto e do tamanho da sua aplicação. Mas pensando nos elementos que geralmente uma página web possui, poderíamos, por exemplo, deixar as partes de uma página (cabeçalho, rodapé, menu lateral) em arquivos distintos.

Você pode ter um arquivo para as famílias de fontes, outro para a paleta de cores. Você pode ter um arquivo com mixins de vendor prefix, um outro arquivo para o reset e muitos outros arquivos.

Vale ressaltar que, com o uso do Sass, todos os arquivos que representam partes serão compilados para um único arquivo. Dessa forma, não temos várias requisições ao servidor. No entanto, para que o Sass saiba da existência de um arquivo particionado, é necessário importá-lo, assunto do próximo tópico.

6.2 IMPORTANDO UMA PARTE DE UM ARQUIVO

Para fazer a importação de um arquivo parte, é usada a diretiva `@import` seguida pelo caminho do arquivo. Note que não é necessário especificar a extensão. Sass é inteligente o suficiente para saber o que fazer.

```
@import "caminhoArquivo";
```

Supondo que fosse necessário importar em um arquivo qualquer os arquivos `_botoes.scss` e `_notificacoes.scss`, o código ficaria:

```
@import "_botoes";  
@import "_notificacoes";
```

Ao efetuar a compilação, note que apenas um arquivo CSS foi gerado. Veja a seguir como fica o diretório onde estão os arquivos.

```
style/
```

```
|
|-- _botoes.scss
|-- _notificacoes.scss
|-- principal.css
|-- principal.scss
```

O CSS seguinte é o resultado da compilação do arquivo principal.scss , no qual foram feitas as importações. Note que tanto os estilos dos botões quanto das notificações estão neste arquivo.

```
.botao, .botaoRedondo, .botaoLargo {
    border: none;
... }
.botao:hover, .botaoRedondo:hover, .botaoLargo:hover {
    background-color: #53BFFF; }

.botaoRedondo {
    border-radius: 4px; }

.botaoLargo {
    height: 40px; }

.alertaErro, .alertaWarning, .alertaSucesso, .alertaInfo {
    width: 80%;
    margin: 20px auto;
    padding: 20px;
...}

.alertaErro {
...}

.alertaWarning {
...}

.alertaSucesso {
...}

.alertaInfo {
...}

/*# sourceMappingURL=principal.css.map */
```

No momento da compilação, o Sass importa as partes. A ordem em que elas estão dispostas é importante, já que as regras CSS podem ser sobrescritas dependendo da ordem em que elas

aparecem.

Vale lembrar de que, quando for necessário usar variáveis ou mixins de outros arquivos, é importante importar o arquivo que os contém antes. Se existir a tentativa de usar uma variável, mixin ou a herança de um seletor não definido, o Sass não conseguirá compilar os arquivos.

O erro exibido a seguir foi gerado em uma tentativa de usar o seletor `.botao` sem que este tenha sido declarado ou importado de outros arquivos: `Change detected to: principal.scss error principal.scss (Line 2: ".botaoErro" failed to @extend ".botao". The selector ".botao" was not found.`

6.3 EXCELENTES CANDIDATOS PARA ARQUIVO PARTICIONADO

Você já ter percebido que a grande maioria das folhas de estilo possui alguns elementos indispensáveis, entre eles estão o reset ou normalize, paleta de cores, família de fontes, partes estruturais de uma página, entre outros. Esses elementos são candidatos fortíssimos a arquivo particionado. Os próximos tópicos mostram alguns exemplos.

Arquivos de reset e normalize

Existem atualmente uma infinidade de navegadores, e cada um deles traz uma estilização padrão para os elementos HTML. Sendo assim, se você utilizar o elemento `h1` em sua página e tentar exibi-la em dois fabricantes distintos de navegadores, será possível notar diferenças sutis na estilização.

Para que o resultado final do trabalho de estilo seja o mesmo, independente do browser usado para a execução da página, é

necessário igualar os estilos padrões aplicados pelos diversos navegadores. Atualmente, os programadores front-end têm usado arquivos de *reset* ou *normalize* para conseguir este resultado.

Quando usado um reset, todas as configurações do navegador são zeradas. Já o *normalize* possui a missão de igualar as propriedades nos diferentes browsers.

Regras de reset podem estar em um arquivo particionado. Existem inúmeros arquivos de reset e normalize espalhados pelos repositórios de código, fóruns e blogs sobre desenvolvimento. Você pode usar um deles, aperfeiçoar algum que você goste ou criar o seu próprio.

Um reset muito utilizado é o do Eric Meyer. Ele possui vários livros e artigos citando inúmeros recursos que podem ser usados nas folhas de estilos, sendo o reset um deles. O trecho a seguir representa um arquivo contendo o reset do Eric Meyer que foi retirado do seu blog pessoal: <http://meyerweb.com/eric/tools/css/reset/>.

```
html, body, div, span, applet, object, iframe, h1, h2, h3, h4, h5,
h6, p, blockquote, pre, a, abbr, acronym, address, big, cite, code
, del, dfn, em, font, img, ins, kbd, q, s, samp, small, strike, st
rong, sub, sup, tt, var, b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend, table, caption, tbody, tfoot, thead
, tr, th, td {
    margin: 0;
    padding: 0;
    border: 0;
    outline: 0;
    font-size: 100%;
    vertical-align: baseline;
}

body {
    line-height: 1;
}
```

Normalmente, os arquivos de reset ou normalize são os

primeiros a serem importados.

Famílias de fontes

No capítulo 2, quando as variáveis foram apresentadas, foi mencionado que um conjunto de variáveis que representam as famílias de fontes poderiam ser separadas em um arquivo. O trecho seguinte representa um arquivo com famílias de fontes definidas como variáveis.

```
$courier: "courier new", courier, freemono, "nimbus mono l","liberation mono", monospace;  
$verdana: verdana, "bitstream vera sans", "dejavu sans","liberation sans", geneva, sans-serif;  
$cambria: cambria, georgia, "bitstream charter","century schoolbook l", "liberation serif", times, serif;  
...
```

Paleta de cores

Também no capítulo 2 foi mencionado que a paleta de cores poderia estar em um arquivo separado. Essa separação ajuda na construção de arquivos pequenos, com um propósito e fáceis de manter.

O trecho de código a seguir possui a definição de alguns itens de uma paleta de cores que pode ficar em um arquivo particionado:

```
...  
$corFundoAtivo: #3baae3;  
$corBordaAtivo: #2694e8;  
$corTextoAtivo: #ffffff;  
$corFundoErro: #cd0a0a;
```

Mixins vendor prefix

Existem alguns recursos do CSS3 que ainda estão em fase de desenvolvimento pelos navegadores, mas que já podemos usar com a inclusão de prefixos que são diferentes para cada fabricante de

browser. Pensando nisso, você pode montar uma biblioteca com esses recursos que você usa constantemente e separá-los em um arquivo como no trecho de código exibido na sequência.

```
@mixin flexbox {
  display: -webkit-box;
  display: -webkit-flex;
  display: -moz-flex;
  display: -ms-flexbox;
  display: flex;
}

@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  -ms-border-radius: $radius;
  border-radius: $radius;
}
```

Nesse código, é possível verificar que foram inclusos os prefixos para o Chrome, Firefox e Internet Explorer. Antes de você montar esse tipo de mixin, é importante que você efetue um levantamento de quais browsers você precisa renderizar a aplicação, para que propriedades desnecessárias não sejam inclusas.

6.4 USANDO PARTICIONAMENTO PARA ARQUITETAR SEU CSS

Quando possuímos um site com muitas páginas ou uma aplicação web com diversos componentes, apenas a separação em arquivos não é suficiente para garantir uma boa estruturação da folha de estilo. Nesses casos, é necessário pensar em uma arquitetura mais robusta na estruturação de arquivos.

Um guia bastante conhecido de como estruturar e modularizar o CSS é o SMACSS. O SMACSS é um documento escrito por Jonathan Snook que contém uma infinidade de dicas de como montar uma estrutura para projetos de todos os tamanhos.

Acompanhe no quadro a seguir um pouco mais sobre o SMACSS.

SMACSS

O SMACSS consiste em uma série de artigos que podem ser acessados em: <http://smacss.com>. Dentre outras ideias, no SMACSS é possível encontrar uma forma de arquitetar o CSS. Em resumo, ele sugere separar as regras de estilo em cinco categorias: base, layout, módulo, estado e tema.

Na base, ficam apenas as regras aplicadas aos seletores que correspondem aos elementos HTML. Isso inclui o arquivo de reset ou normalize e estilizações que devem ser aplicadas nos elementos sem o uso de classes ou IDs. É importante ressaltar que regras aplicadas diretamente nos seletores são globais, o que significa que toda a aplicação deve respeitá-las.

Escreva essas regras para os seletores apenas quando toda a aplicação vai usar. Do contrário, seu CSS ficará cheio de `!important`, o que é uma péssima prática. Sendo assim, para regras não genéricas, dê preferência no uso de classes CSS.

Dentro da pasta `layouts`, devem ficar as regras que correspondem às seções das páginas. Como exemplo, pode ser citado o cabeçalho, o rodapé e o menu lateral da página.

A pasta `módulos` é bem similar a pasta `layouts`. No entanto, deve conter os estilos dos componentes mais discretos da página, como botões, mensagens, tabelas e formulários. Isso significa que, se o componente não for classificado como layout, ele será um módulo.

Já na pasta `states` devem estar as regras que compõem o estado dos componentes quando o usuário estiver navegando

por eles. Por exemplo, a estilização dos comportamentos de `hover` , `selected` e `activated` devem estar nesta pasta. Finalmente, na pasta `temas` , devem estar as estilizações que são diferentes para cada tema da sua aplicação.

Eu, particularmente, gosto da estrutura proposta pelo SMACSS e usei-a como base para uma arquitetura que uso na maioria dos meus projetos. Vale ressaltar que trabalho com softwares do tipo ERP, em que existem infinitas telas, grande quantidades de dados, muitos componentes e muita interferência do usuário no momento da utilização da aplicação.

É importante que você entenda os conceitos e monte uma arquitetura que atenda o contexto em que sua aplicação ou o seu site esteja inserido. Para ajudá-lo nessa tarefa vou apresentar o meu modelo.

Minha arquitetura CSS

```
style/  
|  
|-- base/  
|-- componentes/  
|-- constantes/  
|-- layout/  
|-- modulos/  
|-- paginas/  
|-- vendor/  
|  
|-- style.scss
```

Neste livro, até o presente capítulo, foram apresentadas estilizações de alguns componentes que costumo usar em minhas aplicações. Normalmente, minhas páginas web contêm inúmeros deles e costumo usar uma metodologia própria para efetuar a divisão.

Costumo dividir as minhas folhas de estilo em base, componentes, constantes, módulos, layout, páginas e vendor. A estrutura mostrada anteriormente ilustra a arquitetura que costumo montar para as minhas folhas de estilo.

Constantes

```
style/  
|  
|-- constantes/  
|   |-- _cores.scss  
|   |-- _fontes.scss  
|   |-- _genericos.scss  
|   |-- _main.scss  
|   |-- _prefixos.scss  
|
```

Na pasta `constantes`, costumo guardar tudo o que vai ser reutilizado em todas as demais pastas. Essa estrutura exhibe os arquivos presentes nesta pasta, e também mostra um arquivo chamado `_main`. Este está presente em todas as pastas que compõem a arquitetura e possui como objetivo importar os demais arquivos. Veja o código a seguir:

```
@import "_cores";  
@import "_fontes";  
@import "_prefixos";  
@import "_genericos";
```

O uso do arquivo `_main` em todas as pastas deixa a arquitetura muito mais legível e escalável, pois, no momento da adição ou remoção de um arquivo, apenas a pasta que o contém será alterada.

O arquivo `_cores` contém a paleta de cores da aplicação, e o arquivo `_fontes` contém variáveis com as famílias de fontes. Nesse arquivo, também incluo variáveis para as fontes do tipo `@font-face` que devem ser adicionadas à página.

No arquivo `_prefixos`, coloco mixins que substituem as propriedades ainda em fase de transição pelos navegadores pelos

prefixos correspondentes.

O arquivo `_genericos` contém mixins e placeholders que são usados em lugares distintos da folha de estilo. Como exemplo, posso citar o `truncate` (disponibilizado no capítulo 4) que pode ser usado em muitas partes da aplicação.

A ordem de importação desses arquivos é muito importante, pois não é possível a utilização de uma constante sem esta estar declarada anteriormente. Costumo usar a ordem apresentada no código descrito anteriormente, porque os mixins e placeholders do arquivo `_genericos` podem usar as cores, fontes e prefixos dos demais arquivos da pasta.

Base

```
style/  
|  
|-- base/  
|   |-- _base.scss  
|   |-- _main.scss  
|   |-- _reset.scss  
|
```

Mantenho nas minhas aplicações a pasta `base` que está presente no SMACSS, e costumo utilizá-la para colocar o arquivo de reset e um outro arquivo que também chamo de `_base`, como mostrado na figura anterior.

O arquivo `_base` possui os estilos de seletores naturais do HTML e que devem ser seguidos por toda a aplicação. Neste arquivo, costumo estilizar `body`, `inputs` do tipo texto e radio, `checkbox` e `textArea`. Nas minhas aplicações, esses componentes costumam ter a mesma aparência em toda a aplicação. Por este motivo, deixo as regras na pasta `base`.

Citei o arquivo `_reset` em tópicos anteriores. Eu, particularmente, prefiro usar o reset ao normalize, pois acho melhor

aplicar estilos quando estes estão zerados, do que trabalhar em cima dos estilos do browser. Costumo usar o reset do Eric Meyer com a adição da propriedade `box-sizing`.

Vale lembrar de que a propriedade `box-sizing` com o valor `border-box` calcula a largura do elemento levando em consideração os valores de `padding` e `border`, facilitando bastante os cálculos relativos a tamanho de elementos.

```
@import "_reset";
@import "_base";
```

O arquivo `_main` deve ficar similar ao código mostrado anteriormente. Vale ressaltar que o reset é sempre o primeiro arquivo importado, pois ele possui como objetivo zerar todas as configurações. Sendo assim, não faz muito sentido estilizar algo e depois zerar os valores.

Componentes

```
style/
|
|-- componentes/
|   |-- _botoes.scss
|   |-- _formularios.scss
|   |-- _main.scss
|   |-- _notificacoes.scss
|   |-- _tabelas.scss
|
```

A pasta `componentes` deve conter a estilização de todos os componentes da aplicação como mostra a figura anterior. No SMACSS, a pasta onde as estilizações dos componentes ficam armazenadas é chamada de módulos. No entanto, como geralmente as aplicações maiores contêm componentes e módulos distintos, eu costumo reproduzir a ideia tendo a pasta `componentes` para guardar o estilo do que é componente e a pasta `módulos` para estilizações que devem ser usadas em um módulo específico.

Se a sua folha de estilo possui muitos componentes, é recomendável que você efetue uma subdivisão destes.

Layout

Na pasta `layout`, guardo as regras de estilo das partes que são reutilizáveis na aplicação. No meu caso: cabeçalho, rodapé e uma base para dispor o conteúdo.

Módulos

```
style/
|
|-- modulos/
|   |-- _compras.scss
|   |-- _financeiro.scss
|   |-- _main.scss
|   |-- _rh.scss
|   |-- _vendas.scss
|
```

Como já mencionei, costumo trabalhar com grandes aplicações e estas contêm vários módulos: `financeiro`, `compras`, `rh`, `vendas`, entre outros. Existem algumas regras de estilo que devem ser aplicadas para apenas um módulo destes. Tome como exemplo um estilo de botão que é usado apenas dentro do módulo `financeiro`. Esse estilo, portanto, deve ficar no arquivo `financeiro`, que por sua vez deve ficar na pasta `módulos`, conforme a estrutura mostrada anteriormente.

Dependendo do nível de customizações dos módulos, é possível efetuar uma subdivisão deles.

Páginas

```
style/
|
|-- paginas/
|   |-- _cotacao.scss
```

```
| | -- _login.scss  
| | -- _main.scss  
|
```

Algumas páginas tem um estilo único, ou seja, elas não compartilham as mesmas regras de estilo do restante da aplicação. Costumo estilizar essas páginas separadamente. Como exemplo dessas páginas, posso citar a de login, a de um processo de cotação (na qual uma empresa externa coloca o valor de seus produtos sem ao menos estar logada), ou ainda a página de um processo de recrutamento (em que o candidato insere seus dados).

Vendor

É muito comum usarmos componentes/bibliotecas que foram construídos por outras empresas. Se a aplicação que estamos trabalhando é construída em JavaScript, por exemplo, podemos usar a biblioteca HighCharts (<http://www.highcharts.com/>) para construir os gráficos, ou ainda um calendário do jqueryui (<https://jqueryui.com/datepicker/>).

Ocorre que esses componentes geralmente possuem estilizações que precisam ser adicionadas ao projeto. Costumo colocar no pacote vendor essas regras de estilo que são de componentes de terceiros.

O arquivo style.scss

Por fim, é necessário que tudo seja importado para um único arquivo, e este chamo de `style.scss`. Veja que este arquivo não começa com o caractere underline. Isso significa que ele é o único que será compilado, ou seja, ele deve importar todos os arquivos particionados e transformá-los em um único.

Como já mencionado, todas as pastas devem possuir um arquivo `_main.scss`, no qual todos os demais arquivos da pasta

devem ser importados. Para garantir a compilação de todas as pastas, é necessário que todos os arquivos `_main.scss` de todas elas estejam no arquivo `style.scss`. Veja o trecho de código a seguir:

```
@import "constantes/_main";
@import "base/_main";
@import "componentes/_main";
@import "layout/_main";
@import "modulos/_main";
@import "paginas/_main";
@import "vendor/_main";
```

A ordem de importação é muito importante. Sempre inicio com as constantes, pois elas podem ser usadas em todas as outras pastas. Em seguida, a pasta `base` onde está o reset e a estilização do elementos nativos do HTML. Logo após, adiciono os componentes. As demais pastas podem estar em qualquer ordem, pois raramente as estilizações presentes nelas serão sobrescritas.

6.5 UM PROJETO REAL

Se você está acompanhando a construção do projeto a cada capítulo, já deve ter percebido que a nossa folha de estilo está bem grande e já temos uma certa dificuldade para trabalhar nela. Bom, como neste capítulo vimos como particionar a nossa folha de estilo, vamos aplicar os conceitos em nosso projeto.

Vou começar pela base. Até o momento, não foi adicionado um reset no projeto. Se você renderizar em navegadores diferentes, perceberá uma sutil diferença. Vamos utilizar o mesmo reset já exibido neste capítulo e colocá-lo dentro da pasta `base`.

Na pasta `base` também teremos um arquivo para colocar as estilizações dos seletores naturais do HTML e que devem ser usados por toda a aplicação. Note a seguir que, além dos arquivos `base` e `reset`, temos o arquivo principal que é usado para importar os

demais arquivos. Veja a seguir como fica a estrutura desta primeira pasta.

```
style/  
|  
|-- base/  
|   |-- _base.scss  
|   |-- _principal.scss  
|   |-- _reset.scss  
|  
|-- style.scss
```

Vale lembrar que não vou colocar os trechos de código de cada arquivo aqui, mas você pode consultá-los no repositório de códigos do livro.

Com a base pronta, criarei uma pasta para as constantes. Vou criar três arquivos para armazená-las: um para a paleta de cores, outro para armazenar as variáveis referentes a estilização de fontes e um terceiro para os mixins que podem ser usados em qualquer parte da folha de estilo.

Uma observação importante é que, até o momento, não nos preocupamos com os vendor prefix, dessa forma, o nosso projeto não vai funcionar em browsers mais antigos. Uma ideia é você ir adicionando mixins para eles em um arquivo criado dentro da pasta constantes . A estrutura dessa pasta você pode acompanhar a seguir.

```
style/  
|  
|-- constantes/  
|   |-- _cores.scss  
|   |-- _fontes.scss  
|   |-- _genericos.scss  
|   |-- _principal.scss  
|  
|-- style.scss
```

Agora, vamos separar os estilos do que podemos chamar de componentes. Até o momento, podemos identificar como

componentes os botões, os divisores e ícones. Criamos um arquivo para cada um deles e um arquivo principal para fazer os imports.

```
style/  
|  
|-- componentes/  
|   |-- _botoes.scss  
|   |-- _divisores.scss  
|   |-- _icones.scss  
|   |-- _principal.scss  
|  
|-- style.scss
```

E agora, uma pasta para o layout. Nela particionamos a página em seções para que os arquivos fiquem menores e com poucas estilizações. A estrutura dos arquivos resultantes você pode conferir a seguir.

```
style/  
|  
|-- layout/  
|   |-- _conteudoprincipal.scss  
|   |-- _cabecalho.scss  
|   |-- _feednoticias.scss  
|   |-- _infoconta.scss  
|   |-- _infogerais.scss  
|   |-- _outrasinfo.scss  
|   |-- _sidebar.scss  
|   |-- _boxes.scss  
|   |-- _principal.scss  
|  
|-- style.scss
```

É importante você perceber que, com exceção do arquivo de reset, as demais estilizações já existiam. O que fizemos foi apenas organizá-los em arquivos para facilitar a manutenção.

Vale lembrar que mais divisões podem ser feitas conforme surgir a necessidade. Ou seja, a estilização de outras seções ou de outras páginas podem criar novos arquivos.

6.6 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Neste capítulo, foram apresentadas formas de particionar suas folhas de estilo usando Sass. Algumas ideias e técnicas foram apresentadas para ajudá-lo nessa tarefa.

No capítulo seguinte, você será apresentado ao mundo de funções de cores presentes no Sass. O uso dessas funções lhe ajudará a montar e trabalhar com as paletas de cores de sua aplicação.

USE O PODER DAS CORES

A cor é um dos elementos mais poderosos de um produto web. Isso significa que o uso correto de cores tem um impacto enorme sobre o sucesso do projeto. No entanto, trabalhar com as cores nem sempre é uma tarefa muito fácil. Primeiro você precisa escolher uma cor base para o trabalho e, a partir dela, encontrar outras que deixem a combinação harmônica.

Para ajudá-lo, existem inúmeros sites que facilitam essa tarefa, sugerindo paletas ou esquemas de cores, como o EasyRGB (<http://easyrbg.com/>) e o kuler (<https://color.adobe.com/>).

Imagine que mesmo superando essa dificuldade de busca pelas cores ideais, após dias montando sua obra prima, com uma escolha impressionante de cores, o cliente ou o seu chefe não gostou do resultado e será necessária uma grande mudança. Se você já trabalha na construção de aplicações web, sabe que é extremamente comum esse tipo de mudança, mesmo quando são feitos protótipos antes da construção do produto. E você sabe também que, para fazer uma mudança desse tipo, são necessários tempo e esforço consideráveis.

Bom, o Sass possui uma forma elegante para trabalhar com variações de cores. Em vez de ter uma gama de cores previamente selecionada, é possível escolher apenas uma e criar todas as outras automaticamente usando funções. Isso significa que o Sass disponibiliza funções de cores poderosas o suficiente para permitir que o valor de um tema inteiro de cores seja gerado a partir de uma

ou duas cores de base.

As funções de cores no Sass foram criadas a partir do funcionamento dos sistemas de cores RGB e HSL. Portanto, para compreendê-las, é necessário ter um entendimento de como os sistemas de cores trabalham.

7.1 SISTEMAS DE CORES

Cores em CSS podem ser escritas de muitas maneiras. Talvez a maneira mais simples e primordial de se declarar o valor de uma propriedade de cor é usando o nome da cor. Valores como `green`, `purple`, `gold` e `pink` podem ser setados nas propriedades de cores. No entanto, essa forma é bastante limitada, pois não temos uma grande infinidade de cores disponíveis.

O formato hexadecimal é o sistema de cores mais usado para aplicações web. Ele é composto por seis dígitos, formados por números hexadecimais (na base 16), em que os dois primeiros representam a quantidade de vermelho, os dois seguintes da cor verde e os dois últimos da cor azul. A ausência de cor, ou seja, o valor `#000000`, produz a cor preta. Já a mistura com o valor máximo de todas as cores, `#FFFFFF`, produz a cor branca.

O RGB também é composto pela mistura de vermelho, verde e azul. Para compor a cor, são usados números entre 0 e 255 que representam a quantidade de vermelho, verde e azul que devem ser adicionados para a sua composição.

Do mesmo modo que acontece com o sistema hexadecimal, quando a mistura das três cores está no valor mínimo (0, 0, 0), o resultado é a cor preta. Quando está no máximo (255, 255, 255), resulta na cor branca.

Já o formato HSL é um pouco diferente, pois consiste em um

esquema baseado em um cilindro. Neste, o primeiro valor representa a cor, o segundo a luminosidade, e o terceiro a saturação. Acompanhe a ilustração a seguir.

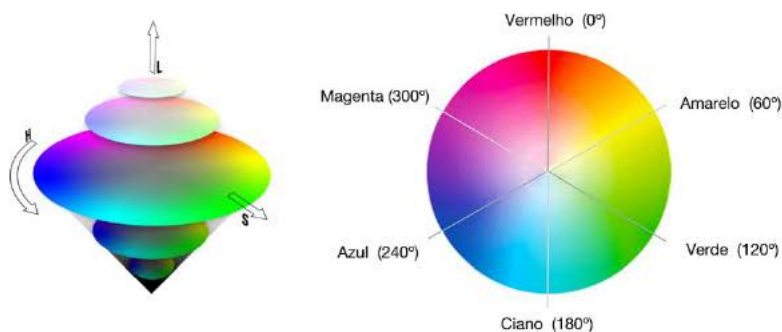


Figura 7.1: <https://goo.gl/4EHafC>

O primeiro valor (hue) é o tom da cor. O seu valor é um número que representa a medida do ângulo onde está o tom da cor no círculo das cores. Alguns valores são:

- 0: vermelho
- 60: amarelo
- 120: verde
- 180: ciano
- 240: azul
- 300: púrpura

O segundo valor é para a saturação (saturation) da cor, ou seja, se ela é viva ou com aspecto desbotado. O seu valor é expresso em percentual. Um valor igual a 100% representa saturação total da cor, e 0 é um leve sombreado cinza de saturação.

O terceiro valor é para a luminosidade (lightness), que equivale ao aspecto claro ou sombrio. O seu valor é expresso em percentual. Um valor igual a 100% resulta em cor branca, e 0 em cor preta,

sendo 50% o valor normal.

Tanto nos formatos RGB quanto HSL podem ser adicionados um quarto valor, que corresponde ao canal alpha, o canal de opacidade. Este último argumento permite indicar a opacidade em uma escala decimal de 0 a 1. Quando adicionado o canal alpha, a combinação nova é chamada de RGBA ou HSLA.

Segue um exemplo com a cor `gold` declarada em diversos formatos. Veja:

```
gold
#ffd700
rgb(255, 215, 0)
hsl(51, 100%, 50%)
```



Figura 7.2: <https://goo.gl/L8xWZm>

7.2 A FUNÇÃO RGBA

Com a função `rgba` do Sass, é possível converter uma cor definida pelo nome ou no formato hexadecimal para um valor `rgba`, incluindo uma camada de transparência. Veja a sintaxe declarada a seguir, na qual os argumentos da função correspondem à cor e à camada de transparência.

Como já mencionado, a cor pode ser definida pelo nome ou no formato hexadecimal, e o canal de transparência deve ser um número entre 0 e 1:

```
rgba(cor, transparencia)
```

Com o intuito de ilustrar a função `rgba`, no exemplo seguinte foi declarada uma variável que representa a cor base. A cor base foi definida usando um valor hexadecimal. Ainda no exemplo apresentado, temos a função `rgba` sendo usada para definir a cor de uma borda. Na função, foram usados como argumento a variável `$corBase` e o valor `.5` como canal de transparência.

```
$corBase: #AD141E;  
background-color: rgba($corBase, .75);  
background-color: rgba($corBase, .5);  
background-color: rgba($corBase, .25);
```

A linha de CSS seguinte representa o código compilado. Note que o Sass transformou o valor hexadecimal em `rgba` e adicionou o valor da camada de transparência.

```
background-color: rgba(173, 20, 30, 0.75);  
background-color: rgba(173, 20, 30, 0.5);  
background-color: rgba(173, 20, 30, 0.25);
```

O resultado a ser renderizado no browser pode ser conferido na figura:



Figura 7.3: <https://goo.gl/X4C8ye>

7.3 RETORNANDO A COMPOSIÇÃO DE UMA COR

O Sass fornece funções capazes de retornar a composição de uma cor. Com elas, é possível saber a quantidade de azul, vermelho e verde que compõe a cor. Também é possível descobrir qual é o seu tom, a luminosidade, a saturação e a opacidade. Veja:

```
green($cor)  
blue($cor)
```



```
red($cor)
hue($cor)
saturation($cor)
lightness($cor)
opacity($cor)
```

Para usá-las, basta indicar a cor cuja composição você deseja saber. Neste momento, você deve estar se perguntando qual seria a real utilidade de saber a quantidade de vermelho que uma cor possui. No próximo capítulo, será abordado como escrever estilos inteligentes, usando lógica de programação, e será possível perceber que o retorno do valor de uma função pode ser usado para modificar a construção de regras de estilo. Por ora, vou dar uma ideia de um possível uso, sem mostrar a sua construção, já que seria necessário usar estruturas lógicas para construir.

Imagine que a grande maioria dos botões da sua aplicação tenha como cor da fonte um `rgb(255, 10, 10)` que é uma variação de vermelho. No entanto, você sabe que alguns botões possuem como cor de fundo também variações de vermelho e, nesses casos, você deseja trocar a cor da fonte para que a usabilidade não fique prejudicada.

Para resolver essa situação de uma forma inteligente, é possível verificar a quantidade de vermelho que a cor de fundo do botão possui, pela função `red`. Considerando que a função pode retornar um número entre 0 e 255 (valores possíveis do RGB), podemos construir uma lógica em que a cor da fonte seja preta sempre que a quantidade de vermelho for maior do que 150. Sei que esse exemplo ficou bem abstrato, porém, algo similar será explorado no próximo capítulo, o que deve clarificar o conceito.

7.4 MIX DE CORES

Com o Sass, é possível misturar duas cores usando a função `mix`. Para usar a função `mix`, você deve usar duas cores separadas

por vírgula. A variável peso é opcional. Se você não indicar o peso, a mistura será efetuada com 50% de cada cor.

Caso queira uma mistura com percentuais diferentes, você deve indicar o percentual da primeira cor. O código a seguir apresenta a sintaxe.

```
mix(primeiraCor, segundaCor, peso)
```

O exemplo de mix de cores apresentado a seguir utiliza duas cores que estão representadas pelas variáveis `$corBase` e `$corSecundaria`. Note que o primeiro mix foi feito sem indicar o peso, portanto, a mistura efetuada terá 50% de cada cor. O segundo mix indica 70% de peso, o que significa que a mistura terá 70% da `$corBase` e 30% da `$corSecundaria`.

```
$corBase: #AD141E;  
$corSecundaria: #0750E1;  
background-color: mix($corBase, $corSecundaria);  
background-color: mix($corBase, $corSecundaria, 70%);
```

Note no CSS gerado, exibido a seguir, que o Sass mesclou as duas cores, deixando-as no formato hexadecimal. Você também pode usar o formato de cores RGBA como argumento da função `mix`. Nesse caso, o resultado será em RGBA.

```
background-color: #5a3280;  
background-color: #7b2659;
```

A figura seguinte exibe o resultado do mix de cores no browser.



Figura 7.4: <https://goo.gl/jsNni5>

7.5 ESCURECENDO E CLAREANDO CORES

No Sass, as funções `lighten` e `darken` servem

respectivamente para clarear e escurecer uma cor. Ambas trabalham com um valor correspondente ao percentual de clareamento ou escurecimento da cor. Veja:

```
lighten (cor, percentualClareamento);  
darken (cor, percentualEscurecimento);
```

O trecho de código a seguir usa os valores de 10%, 20% e 30% de clareamento e escurecimento da cor representada pela variável `$corBase`.

```
$corBase: #AD141E;  
.lighten10{  
    background-color: lighten($corBase, 10%);  
}  
.lighten20{  
    background-color: lighten($corBase, 20%);  
}  
.lighten30{  
    background-color: lighten($corBase, 30%);  
}  
  
.darken10{  
    background-color: darken($corBase, 10%);  
}  
.darken20{  
    background-color: darken($corBase, 20%);  
}  
.darken30{  
    background-color: darken($corBase, 30%);  
}
```

Neste caso, o CSS compilado possui a definição das cores no formato hexadecimal, pois foi o formato usado como valor de entrada da função. Como já mencionado, os demais formatos permitidos pelo Sass poderiam ser usados como valor de entrada.

```
.lighten10 {  
    background-color: #db1926; }  
  
.lighten20 {  
    background-color: #e93e49; }  
  
.lighten30 {  
    background-color: #ee6c74; }
```

```
.darken10 {
  background-color: #7f0f16; }

.darken20 {
  background-color: #52090e; }

.darken30 {
  background-color: #240406; }
```

Note nas figuras o resultado das funções apresentadas neste tópico no navegador.

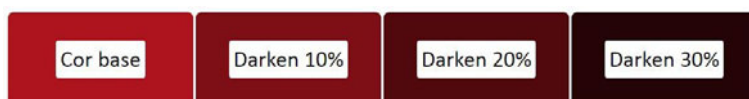


Figura 7.6: <https://goo.gl/i8vIkW>



Figura 7.5: <https://goo.gl/UCg1XF>

7.6 CRIANDO CORES INVERSAS E COMPLEMENTARES

As cores complementares são aquelas que mais oferecem contraste entre si. Elas ocupam lugares opostos no círculo cromático, ou seja, a cor a 180 graus. Veja a imagem a seguir.

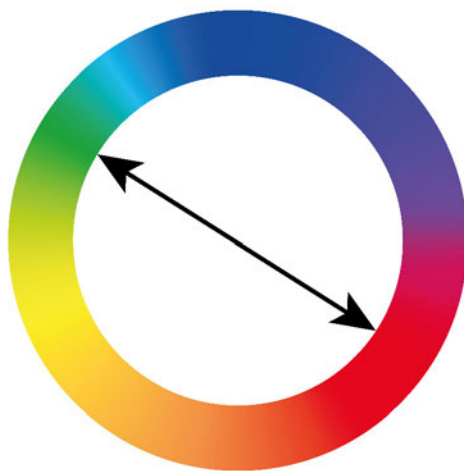


Figura 7.7: <https://goo.gl/WEXQaV>

O Sass possui a função `complement` que retorna a cor complementar. Esta função necessita apenas de um argumento: a cor cujo complemento você quer.

A função `invert` retorna o negativo de uma cor. Ela é muito semelhante à função `complement`, com uma sutil diferença: os valores de azul, vermelho e verde são invertidos e a opacidade permanece como está.

O trecho de código adiante demonstra a utilização das funções `invert` e `complement`. Note que foi usada a mesma cor base para as duas funções. Dessa forma, é possível comparar a sutil diferença entre os dois resultados de renderização no browser.

```
$corBase: #AD141E;
.invert{
  background-color: invert($corBase);
}
.complement{
  background-color: complement($corBase);
}
```

Acompanhe o trecho de código e o resultado renderizado no browser.



Figura 7.8: <https://goo.gl/m5viSt>

7.7 SATURAR, DESSATURAR

A saturação é a intensidade da cor. Isso significa que quanto mais alta é a saturação de uma cor, mais viva e brilhante ela é. O contrário, imagens com baixa saturação tendem a ser próximas do cinza. O Sass possui funções para saturar e dessaturar uma cor.

A função `saturate` deixa a cor mais saturada. É necessário indicar para a função a cor e quantidade de saturação a ser incrementada que pode ser um valor entre 0% e 100%. Já a função `desaturate` deixa a cor menos saturada.

Também é necessário indicar a cor e um valor entre 0% e 100% que indica a quantidade de dessaturação a ser aplicada. Acompanhe o trecho de código a seguir:

```
$corBase: #AD141E;
$corBaseSat: #655C5D;

.saturate25{
  background-color: saturate($corBaseSat, 25%);
}
.saturate50{
  background-color: saturate($corBaseSat, 50%);
}
.saturate75{
  background-color: saturate($corBaseSat, 75%);
}
.desaturate25{
  background-color: desaturate($corBase, 25%);
}
```

```
.desaturate50{
    background-color: desaturate($corBase, 50%);
}
.desaturate75{
    background-color: desaturate($corBase, 75%);
}
```

No código apresentado foi feita a saturação e a dessaturação de cores nos níveis de 25%, 50% e 75%. Acompanhe adiante o resultado renderizado no browser.

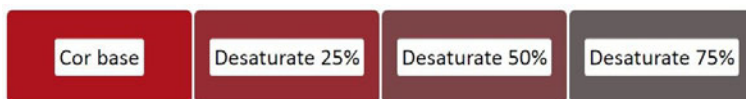


Figura 7.10: <https://goo.gl/AFNNrH>



Figura 7.9: <https://goo.gl/ym0uJv>

7.8 AJUSTAR O MATIZ DE UMA COR

A função `adjust-hue` é capaz de produzir o mesmo efeito da função `complement`, usando o valor de 180 deg. Entretanto, com `adjust-hue`, é possível criar uma gama imensa de cores variando o grau de coloração. Isso pode ser feito com valores positivos ou negativos no intervalo de 0 a 360.

Acompanhe no trecho de código a seguir, o uso da função `adjust-hue` para uma determinada cor para os ângulos 45, 90 e 180.

```
$corBase: #AD141E;
.adjusthue45{
    background-color: adjust-hue($corBase, 45deg);
}
.adjusthue90{
    background-color: adjust-hue($corBase, 90deg);
}
```

```

}
.adjusthue180{
    background-color: adjust-hue($corBase, 180deg);
}

```

O resultado renderizado no browser deve ser similar à figura a seguir. Nele é possível ver a grande diferença entre os tons das cores apresentados. Também é possível verificar que, quando o valor do ângulo é 180, o resultado é idêntico ao uso da função `complement`.



Figura 7.11: <https://goo.gl/aLt6KM>

7.9 ADICIONANDO CANAL ALPHA

Em tópicos anteriores, foi apresentada a função `rgba` que adicionava um canal de transparência a uma cor. É possível obter o mesmo efeito com o uso da função `transparentize` ou `fade-out`.

A função `transparentize` espera uma cor e um valor entre 0 (totalmente transparente) e 1 (totalmente opaco) para definir a transparência. Já a função `fade-out` trabalha da mesma forma, com a diferença que ela aceita uma cor no formato HSL. Veja o trecho de código a seguir:

```

$corBase: #AD141E;
.transparentize30{
    background-color: transparentize($corBase, .3);
}
.transparentize60{
    background-color: transparentize($corBase, .6);
}
.transparentize90{
    background-color: transparentize($corBase, .9);
}
.fadeout30{
    background-color: fade-out($corBase, .3);
}

```



```

}
.fadeout60{
  background-color: fade-out($corBase, .6);
}
.fadeout90{
  background-color: fade-out($corBase, .9);
}

```

Com uma análise no trecho de código apresentado, é possível perceber que as funções `transparentize` e `fade-out` receberam os mesmos argumentos a fim de demonstrar a semelhança entre elas. Note que, em ambos os casos, foram adicionados canais de transparência de 0.3, 0.6 e 0.9. A imagem a seguir ilustra o resultado no browser.

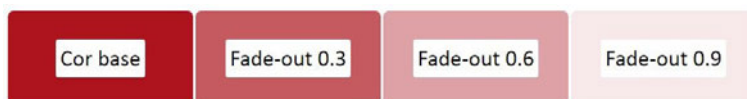


Figura 7.13: <https://goo.gl/WIY9Oo>

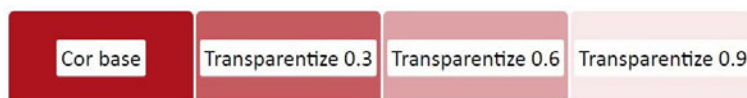


Figura 7.12: <https://goo.gl/o0rRI2>

7.10 EM TONS DE CINZA

Sass possui a função `grayscale` que converte a cor passada para um tom de cinza. Para usá-la, basta informar o valor da cor. Veja:

```

$corBase: #AD141E;
.grayscale{
  background-color: grayscale($corBase);
}

```

O resultado na renderização no browser deve ficar similar à figura apresentada adiante.



Figura 7.14: <https://goo.gl/aLFsAo>

7.11 UM PROJETO REAL

Continuando o nosso projeto, vamos aproveitar as funções de cores para criar vários temas para a nossa aplicação. Veja a figura:

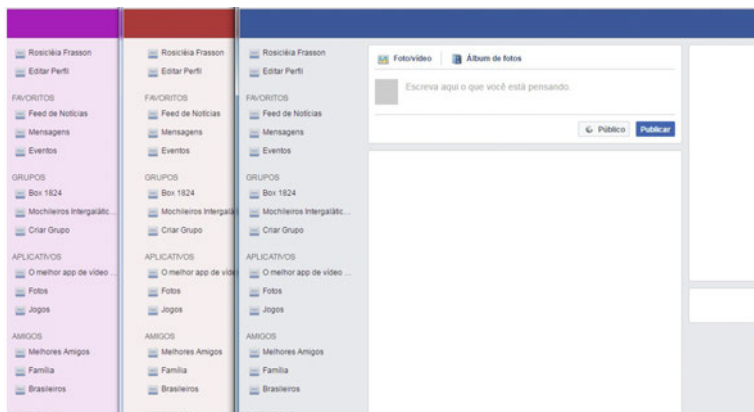


Figura 7.15: <https://goo.gl/bVLuOm>

Para conseguir o efeito esperado, vamos fazer algumas alterações na estrutura do nosso código. A primeira delas é transformar a cor da borda do cabeçalho em uma variável, como exibido no trecho de código seguinte. Lembre de transferir essa variável para o arquivo de cores já existente.

```
$corBordaPrincipal: #29487D;
.cabecalho{
  height: 45px;
  background-color: $corBase;
  border-bottom: 1px solid $corBordaPrincipal;
}
```

Na imagem que aparece no início do tópico, é possível verificar que algumas cores variam de acordo com o tema e outras permanecem iguais, como é o caso dos divisores, o botão padrão e a grande maioria das cores de fonte. Sendo assim, vamos separar nossas cores em dois blocos: o primeiro representa as cores que permanecem iguais para todos os temas, e o segundo compreende as cores que podem ser alteradas para cada tema. Acompanhe a divisão:

```
$corFontePrincipal: #1D2129;
$corFonteDestaque:#FFFFFF;
$corDivisor: #CCC;
$corFundoBoxes: #FFFFFF;
$botaoPadrao: #F6F7F9;
$botaoPadraoHover: #E9EBEE;
$botaoPadraoBorda: #CED0D4;
$botaoPadraoFonte: #4B4F56;

//Cores que podem variar de acordo com o tema
$corBase: #3B5998;
$corBordaPrincipal: #29487D;
$corHoverPrincipal: #DDDFE2;
$corFonteSecundaria: #365899;
$corFundo: #E9EBEE;
$botaoDestaque: #4267B2;
$botaoDestaqueHover: #365899;
$botaoDestaqueBorda: #4267B2;
$botaoDestaqueFonte: #FFFFFF;
```

Agora vamos fazer uso das funções de cores do Sass para facilitar a criação dos vários temas. Veja no código seguinte que podemos usar uma função de cor ou a combinação de mais de uma delas, para se chegar à cor desejada.

```
$corBase: #3B5998;
$corBordaPrincipal: darken( $corBase, 8%);
$corHoverPrincipal: lighten(desaturate($corBase, 36), 46%);
$corFonteSecundaria: darken($corBase, 10%);
$corFundo: lighten(desaturate($corBase, 30),50%);
$botaoDestaque: lighten($corBase, 6%);
$botaoDestaqueHover: darken( $corBase, 10%);
$botaoDestaqueBorda: lighten($corBase, 7%);
```

Vale ressaltar que as cores vão ficar similares as que existiam no projeto original. Com o agrupamento de várias funções de cores, é possível chegar a qualquer cor desejada. Mas para que nosso projeto não fique muito complexo, limitei ao uso de no máximo duas funções para a construção de cada cor.

A imagem a seguir mostra como deve ficar a página com o uso das funções de cores. Note que a diferença nas cores é praticamente imperceptível.

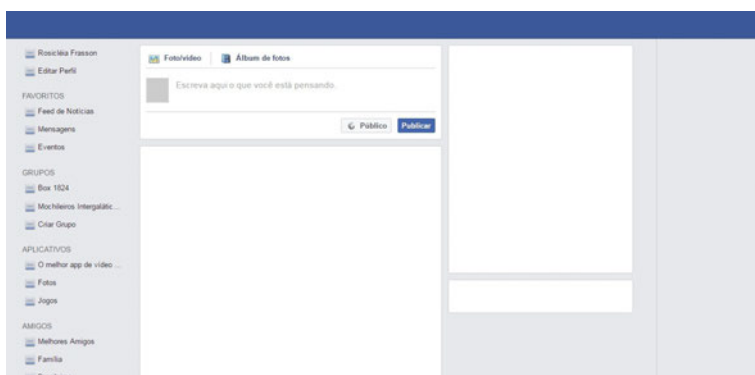


Figura 7.16: <https://goo.gl/mC53K4>

E para mudarmos o tema, basta alterar a cor base. Veja:

`$corBase: #A41EB7;`

E o resultado renderizado é o da figura seguinte.

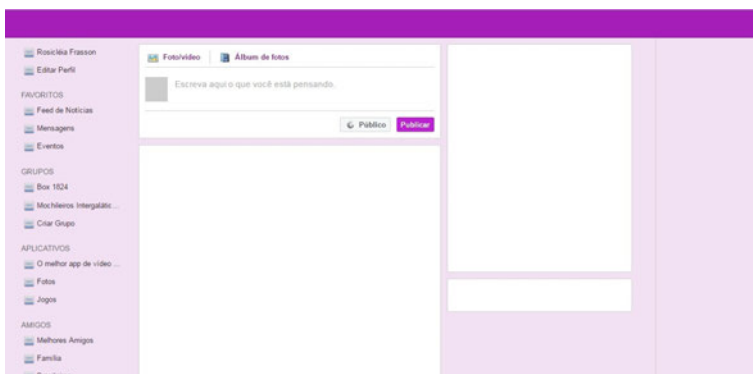


Figura 7.17: <https://goo.gl/8KF7Tz>

E se trocarmos a base novamente:

`$corBase: #AA3939;`

O resultado é este:

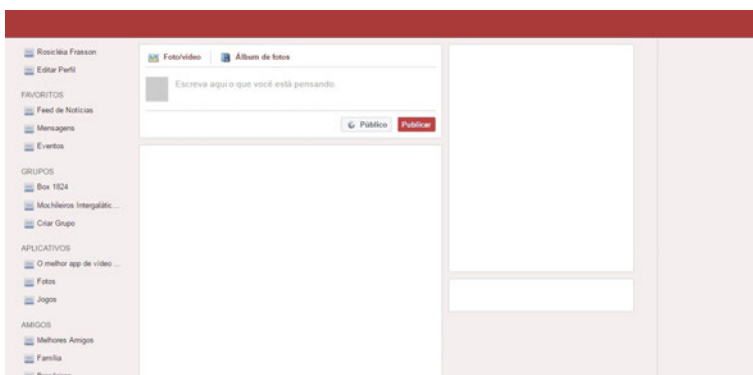


Figura 7.18: <https://goo.gl/HieUAR>

Simplesmente fantástico, não é mesmo?

Lembre-se de que, para ficar mais harmônico, você também pode substituir as cores que deixamos fixas por funções de cores. Você pode também usar uma combinação maior de funções para uma variação maior de cores. Enfim, você pode tudo.

7.12 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Neste capítulo, foram apresentadas as diversas funções de cores presentes no Sass e o efeito alcançado com o uso de cada uma delas. Também montamos alguns temas para o nosso projeto usando as funções apresentadas.

No capítulo seguinte, vamos mergulhar de cabeça nos tipos de dados e as operações suportadas. Isso será fundamental para a criação de estilos muito mais inteligentes.

DESVENDANDO E MANIPULANDO OS TIPOS DE DADOS

Um tipo de dado é uma forma de classificar um tipo de informação. Todo valor de uma propriedade CSS ou uma variável Sass possui um tipo. Como tipos de dados suportados pelo Sass, podem ser citados os números, as strings, as cores, os valores booleanos, as listas e mais recentemente os mapas.

A grande maioria desses tipos de dados é familiar para quem conhece CSS e já foram usados nos capítulos anteriores. Neste capítulo, os tipos de dados serão abordados em detalhes.

Vamos também conhecer as operações suportadas pelos tipos de dados. As operações consistem em um trecho de código, previamente escrito, que retorna algum valor. Elas podem ser extremamente úteis para a escrita de folhas de estilo. Com as operações numéricas, por exemplo, é possível, construir layouts fluidos com mais facilidades.

Um outro recurso importante que será visto é a interpolação. Com ele, além de outras funcionalidades, é possível construir um mixin genérico que adicione qualquer tipo de vendor prefix a sua folha de estilo. Interessante, não é mesmo?

8.1 NUMBERS

O CSS faz uso pesado de valores numéricos para a construção de layouts, por este motivo é muito importante entender o seu funcionamento. No Sass, como no CSS, um número possui duas partes: um valor numérico e opcionalmente uma unidade. É importante ressaltar que o valor 0 normalmente não acompanha uma unidade.

Como exemplo de valores numéricos suportados pelo Sass podemos citar 10px, 2.5, 5em, 45 e 3%. O Sass disponibiliza algumas funções numéricas:

- `round($numero)` — Arredonda um número para o número inteiro mais próximo.
- `ceil($numero)` — Arredonda um número para o próximo número inteiro.
- `floor($numero)` — Arredonda um número para o número inteiro anterior.
- `abs($number)` — Retorna o valor absoluto de um número.
- `min($numeros...)` — Encontra o menor valor entre vários números. Essa função pode receber qualquer quantidade de argumentos.
- `max($numeros...)` — Encontra o maior valor entre vários números. Essa função pode receber qualquer quantidade de argumentos.
- `random($limite)/random()` — Retorna um número gerado entre o intervalo mencionado. Você pode usar a função com um valor limite. Nesse caso, o número gerado deve estar entre 0 e o número limite. Como pode usar sem argumento, sendo que, nesse caso, o número gerado estará entre 0 e 1.

8.2 STRINGS

As strings são o tipo de dado mais usado no CSS. Elas são constituídas por uma cadeia de caracteres que pode ser delimitada por aspas simples ou duplas, ou palavras reservadas para valores do CSS, como `bold`, `center`, `border-box`.

O operador `+`, usado quando um dos operadores é uma string, resulta em uma concatenação. Isso significa que adicionando uma string com outro valor, string ou não, o resultado é uma nova string com os valores concatenados. Veja:

```
$testeConcatenacao = "teste" + 1; //teste1
```

Com o intuito de facilitar a manipulação de strings, algumas funções são disponibilizadas pelo Sass. Veja:

- `str-length($string)` — Retorna a quantidade de caracteres presente em uma string.
- `str-index($string, $substring)` — Retorna o índice da primeira ocorrência da `$substring` na `$string`.
- `str-slice($string, $inicio, $fim)` — Extrai uma substring de uma string.
- `to-upper-case($string)` — Coloca todos os caracteres da string em caixa alta.
- `to-lower-case($string)` — Coloca todos os caracteres da string em caixa baixa.

8.3 BOOLEANS

Booleano é um tipo de dado que permite apenas dois valores: verdadeiro (`true`) ou falso (`false`). Eles são usados para a tomada de decisões sobre quais estilos usar. Normalmente, eles resultam de operações relacionais ou lógicas que são usadas nas

diretivas `@if` e `@while` que serão apresentadas no capítulo 9.

8.4 LISTS

As listas são similares aos vetores, encontrados na ampla maioria das linguagens de programação. Como o próprio nome sugere, um dado do tipo `list` constitui em uma listagem de valores.

As listas podem armazenar valores de qualquer tipo como números, cores, strings e listas — neste caso, temos listas aninhadas. Para efetuar a declaração de um tipo lista, basta separar os valores por espaços ou vírgulas. Acompanhe a seguir duas listas compostas por strings, em que uma é separada por espaço e a outra por vírgula.

```
$listaEspaco: "item-1" "item-2" "item-3";  
$listaVirgula: "item-1", "item-2", "item-3";
```

Com Sass, também é possível criar listas aninhadas, você pode usar qualquer nível de profundidade que desejar. Para isso, é necessário, usar parênteses ou tipos distintos de separadores para delimitar. Veja no exemplo adiante que, na primeira lista, são usados parênteses e vírgulas como separadores, já na segunda lista são usados espaços e vírgulas:

```
$listaParenteses: (  
  ("item-1.1", "item-1.2", "item-1.3"),  
  ("item-2.1", "item-2.2", "item-2.3"),  
  ("item-3.1", "item-3.2", "item-3.3")  
);  
  
$lista: "item-1.1" "item-1.2" "item-1.3",  
  "item-2.1" "item-2.2" "item-2.3",  
  "item-3.1" "item-3.2" "item-3.3";
```

É importante ressaltar que normalmente são usados parênteses para delimitar uma lista. Isso aumenta a legibilidade. Entretanto, no Sass, seu uso é opcional, exceto quando você precisa criar uma lista vazia.

```
$list: (); // Lista vazia
```

Um outro ponto bem relevante é que os índices das listas no Sass iniciam em 1 e não em 0, como na maioria das linguagens de programação. Veja a figura seguinte, que representa a estrutura de uma lista no Sass, na qual os valores que representam uma paleta de cores estão armazenados em uma variável chamada `$paletaCores` do tipo lista.

	1	2	3	4	5	6
\$paletaCores:	#deedf7	#aed0ea	#222222	#f2f5f7	#dddddd	#e4f1fb

Figura 8.1: <https://goo.gl/M61tmr>

Para trabalhar com as listas, o Sass possui uma série de funções. Veja:

- `length($lista)` — Retorna a quantidade de itens de uma lista.
- `append($lista, $valor)` — Adiciona um item na lista com o `$valor` passado como parâmetro.
- `set-nth($list, $indice, $valor)` — Altera o valor do item da lista no índice passado.
- `nth($lista, $indice)` — Retorna o valor na posição do `$indice` na lista. Se for passado um índice não existente na lista, um erro de índice inexistente é gerado. Se você usar como valor do índice um número negativo, a busca pelo elemento é feita na ordem inversa, ou seja, a partir do fim da lista.
- `index($lista, $valor)` — Retorna o primeiro índice do `$valor`, ou `null` caso a lista não contenha nenhum item com o dado valor.

8.5 MAPS

O tipo de dados chamado maps foi incluído no Sass na versão 3.3. É um tipo de dado já existente nas linguagens de programação, como JavaScript, e constitui um conjunto de pares com chave e valor.

No Sass, um mapa usa parênteses como delimitadores externos, dois pontos para mapear as chaves e valores, e vírgulas para separar os pares de chave/valor. O trecho de código seguinte mostra um map válido para o Sass. Se você conhece o formato JSON, vai perceber que é similar.

```
$mapa: (  
  chave: valor,  
  outraChave: outroValor  
);
```

Vale ressaltar que o uso da vírgula no último par chave/valor é opcional. Um outro ponto importante é que as chaves devem ser únicas, ou seja, não é permitido duas chaves iguais.

Um bom uso de mapas é para definição da paleta de cores. Embora você possa usar variáveis distintas para armazenar as cores, pode ficar muito confuso se você possui dezenas delas. Veja o exemplo a seguir:

```
$paletaCores:(  
  fundoDestaque: #deedf7,  
  bordaDestaque: #aed0ea,  
  textoDestaque: #222222,  
  fundoConteudo: #f2f5f7,  
  bordaConteudo: #dddddd,  
  fundoHover: #e4f1fb,  
  bordaHover: #74b2e2,  
  textoHover: #0070a3  
);  
  
.botao {  
  background-color: map-get($paletaCores, fundoDestaque);  
}
```

Note que o mapa foi usado para armazenar a paleta de cores,

usando pares de chave e valor. É possível também observar que o seletor `.botao` acessa uma das cores da paleta através da função `map-get`, disponibilizada pelo Sass. Outras funções também estão disponíveis. Acompanhe:

- `map-get($mapa, $chave)` — Retorna o valor no mapa associado com a chave.
- `map-remove($mapa, $chave)` — Remove o par, chave e valor associado a chave.
- `map-keys($mapa)` — Retorna uma lista com todas as chaves existentes no mapa.
- `map-values($mapa)` — Retorna uma lista com todos os valores do mapa.

8.6 OPERAÇÕES

Operadores aritméticos

O Sass suporta basicamente cinco operações matemáticas: adição (`+`), subtração (`-`), divisão (`/`), multiplicação (`*`) e módulo (`%`). O operador módulo representa o resto de uma divisão inteira. Os demais operadores representam as operações básicas da matemática e dispensam apresentações.

O trecho de código exibido a seguir representa as operações citadas. Observe que foram usadas como nome das variáveis que armazenam os valores das operações com os mesmos nomes que elas representam apenas para facilitar a legibilidade do exemplo. Na prática, o nome da variável pode ser qualquer um. Também com o intuito de facilitar o entendimento, foi colocado como comentário o resultado da expressão.

```
$adicao: 4 + 2; // 6
$subtracao: 4 - 2; // 2
$multiplicacao: 4 * 2; // 8
```



```
$divisao: 4 / 2; // 2  
$resto: 4 % 2; // 0
```

Embora não seja obrigatório, é fortemente recomendado que você utilize espaços entre os operandos e operadores. Isso facilita bastante a legibilidade.

A precedência de operadores segue as mesmas regras da álgebra. Isso significa que você pode forçar com o uso de parênteses que operações sejam prioritariamente executadas.

As operações com números acompanhados por unidades merecem uma atenção especial, pois algumas unidades são compatíveis e outras não. Sass costuma preservar as unidades durante a execução de operações aritméticas. No entanto, existem algumas incompatibilidades entre as unidades. Acompanhe:

- px , in , cm , mm , pc e pt são compatíveis.
- deg , rad , grad e turn são compatíveis
- s e ms são compatíveis
- Hz e kHz são compatíveis
- dpi , dppx e dpcm são compatíveis

Temos também operações com a mesma unidade que não podem ser executadas, como a multiplicação de dois valores com a unidade px. Essa operação resultaria em uma unidade quadrática, que não é suportada pelo CSS.

Operadores relacionais

Os operadores relacionais suportados pelo Sass são igual (==), maior (>), maior ou igual (>=), menor (<), menor ou igual (<=) e diferente (!=). Estes operadores também funcionam da mesma maneira já conhecida na álgebra. Acompanhe a seguir o resultado do uso desses operadores.

```
$a: 4 < 2; // false
```

```
$a: 4 <= 2; // false
$a: 4 == 2; // false
$a: 4 => 2; // true
$a: 4 > 2; // true
$a: 4 != 2; // true
```

Operadores lógicos

Operadores lógicos permitem testar múltiplas condições usando uma expressão condicional. O Sass possui os operadores lógicos `and`, `or` e `not`. Veja como eles funcionam:

- `and` — Retorna verdadeiro se ambas as variáveis de entrada forem verdadeiras.
- `or` — Retorna verdadeiro se ao menos uma das variáveis de entrada for verdadeira.
- `not` — Representa a negação (inverso) da variável atual. Se ela for verdadeira, torna-se falsa, e vice-versa.

Geralmente, os operadores lógicos são usados em conjunto com as estruturas condicionais que serão abordadas no capítulo 9.

Interpolação

A interpolação é um mecanismo que pode ser usado em conjunto com os mixins para ampliar seu poder de reutilização. Consiste no processo de inserção do valor de uma variável em outras partes do Sass. Ou seja, expressões ou strings contendo uma ou mais variáveis são avaliadas, produzindo um resultado em que as variáveis são substituídas por seus valores correspondentes na memória.

A utilização de `#{ }`, envolvendo uma expressão, em um seletor ou nomes de propriedades caracteriza uma interpolação.

O exemplo apresentado a seguir possui um mixin responsável por adicionar uma borda. Note que o lado em que a borda é

adicionada é um valor definido no uso do mixin. Dessa forma, é possível que o mesmo mixin adicione borda superior, inferior, do lado direito ou esquerdo. Também é possível perceber que, no exemplo, a interpolação foi usada para construir o nome da propriedade como `border-top`.

```
@mixin set-border($lado) {  
    border-#{ $lado }: solid 1px #000;  
}  
.header {  
    @include set-border("top");  
}
```

Quando compilado para CSS, este código deve gerar o trecho de estilo apresentado a seguir. Note que `{ $lado }` foi substituído por `top`.

```
.header {  
    border-top: solid 1px #000;  
}
```

Com o uso da interpolação, também é possível criar um mixin genérico para atender ao vendor prefix dos browsers. Veja o código adiante:

```
@mixin prefixos($propriedade, $valor) {  
    -moz-#{ $propriedade }: $valor;  
    -webkit-#{ $propriedade }: $valor;  
    -ms-#{ $propriedade }: $valor;  
    #{ $propriedade }: $valor;  
}  
.botao{  
    @include prefixos (border-radius, 10px);  
}
```

Note que, com o código apresentado, pode ser criado vendor prefix para uma infinidade de propriedades. Neste exemplo, foi usado o `border-radius`, porém, poderíamos usar o mesmo mixin para qualquer outra propriedade que necessite de prefixos para funcionar em determinados browsers. O código compilado deve ficar similar ao apresentado a seguir:


```
.botao{  
  -moz-border-radius: 10px;  
  -webkit-border-radius: 10px;  
  -ms-border-radius: 10px;  
  border-radius: 10px;  
}
```

8.7 FUNÇÕES

No capítulo anterior e também em alguns tópicos deste capítulo, você foi apresentado a uma série de funções disponíveis pelo Sass, como as funções para a manipulação de cores, as funções numéricas, função para acesso de strings e listas. No entanto, você já deve ter percebido que nem todas as funções de que você pode precisar estão disponíveis na biblioteca e talvez você goste da ideia de construir as suas próprias funções. Com o Sass, isso é possível.

Imagine que você costuma usar um cálculo matemático para montar seu layout, ou você pode gostar de misturar várias funções de cores para ter um efeito diferente: essas e muitas outras coisas podem se tornar uma função.

Em computação, uma função é um trecho de código que retorna um resultado. Podemos dizer que é a maneira para extrair trechos de códigos repetidos e transformá-los em um trecho de código reutilizável.

No Sass, em termos de construção, uma função é muito similar a um mixin. No entanto, uma função é concebida com o intuito de retornar um valor que pode ser qualquer tipo de dado. Já um mixin tem como objetivo gerar estilos.

No Sass, para construir uma função, é necessário usar a diretiva `@function` seguida pelo nome da função. Em seguida, é necessário colocar um par de parênteses — possivelmente, mas não necessariamente, contendo parâmetros passados para a função.

Como nos mixins, os limites do bloco referente a função, ou seja, seu início e o fim, ficam entre chaves, e cada instrução é finalizada com um ponto e vírgula.

As funções sempre devem retornar um valor, que pode ser qualquer tipo de dado suportado pelo Sass. Para definir o valor retornado pela função, usamos a diretiva `@return`. O trecho de código a seguir ilustra como uma função deve ser declarada.

```
@function nomeFuncao($argumento1, $argumento2){  
  @return $argumento1 + $argumento2;  
}
```

Uma função é chamada pelo seu nome e sua lista de argumentos. A lista de argumentos deve estar na mesma ordem em que está na declaração da função. Veja:

```
h1{  
  padding: nomeFuncao(10px, 30px);  
}
```

Ou você pode usar palavras chaves. Dessa forma, os argumentos podem estar em qualquer ordem. Acompanhe no exemplo a seguir.

```
h1{  
  padding: nomeFuncao($argumento2:30px, $argumento1:10px);  
}
```

Também é possível usar valores padrões na declaração das funções. Dessa forma, no momento de usá-la, o valor para a variável é opcional. Veja no exemplo a seguir que o `argumento1` já possui 15px como valor default. No momento de invocar essa função, você pode usar um outro valor, ou deixar que a função seja executada com 15px.

```
@function nomeFuncao($argumento1: 15px, $argumento2){  
  @return $argumento1 + $argumento2;  
}
```

É importante ressaltar que uma função pode ser usada para

definir o valor de uma variável como valor para uma propriedade de algum seletor, ou como valor de argumento para um mixin ou outra função.

8.8 UM PROJETO REAL

Neste capítulo, vamos aproveitar o conhecimento sobre os maps do Sass para adicionar um box de reações. O mapa será utilizado para armazenar as imagens que são usadas para representar cada reação. A imagem seguinte é uma prévia do que vamos construir nos próximos parágrafos.

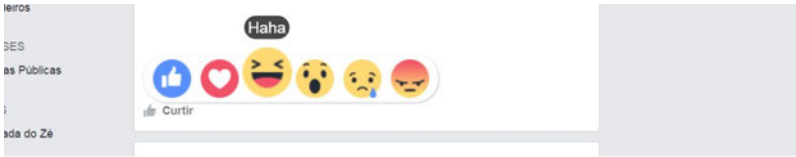


Figura 8.2: <https://goo.gl/Nt4PYz>

Primeiramente, vamos inserir o HTML responsável pela construção do reactions na página. Vamos adicionar o trecho de código a seguir dentro de um dos box de publicação.

```
<div class="acoes">
  <div class="curtir">
    <span class="btCurtir">Curtir</span>
    <div class="reacoes">
      <div class="reacaoCurti"><span>Curti</span></div>
      <div class="reacaoAmei"><span>Amei</span></div>
      <div class="reacaoHaha"><span>Haha</span></div>
      <div class="reacaoUau"><span>Uau</span></div>
      <div class="reacaoTriste"><span>Triste</span></div>
      <div class="reacaoGrr"><span>Grr</span></div>
    </div>
  </div>
</div>
```

Enquanto outros conteúdos não forem adicionados ao bloco publicação, vamos adicionar como posição padrão para os

elementos a parte inferior. Veja:

```
.boxPublicacao{  
  height: 500px;  
  display: flex;  
  align-items: flex-end;  
}
```

Se você renderizar a página, deve ter a sua listagem de reações como na figura a seguir.



Figura 8.3: <https://goo.gl/P1qsXG>

Como você pode perceber, vamos precisar de um pouquinho de estilo para melhorar a aparência das nossas reações. Vamos começar criando um arquivo dentro da pasta `componentes` para estilizar as ações e reações.

Inicialmente, vamos estilizar o botão curtir. Você pode acompanhar no trecho de código a seguir que apenas estilizações simples de cor, tamanho de fonte e espaçamento foram feitas.

```
.btCurtir{  
  position: relative;  
  color: #7f7f7f;  
  display: inline-block;  
  font-size: 12px;  
  font-weight: bold;  
  line-height: 14px;  
  margin-right: 20px;  
  padding: 4px 4px 4px 0;  
  cursor: pointer;  
}
```

Continuando nossa estilização, o botão curtir possui um ícone.

Precisamos adicioná-lo. Vamos usar a técnica do `content` vazio para esse fim. Acompanhe como deve ficar o trecho de estilo.

```
.btCurtir{
  ...
  &:before{
    background-image: url(img/curtir.png);
    background-repeat: no-repeat;
    content: '';
    display: inline-block;
    margin: 0 6px -3px 0;
    width: 14px;
    height: 14px;
  }
}
```

Renderizando a página neste momento, você poderá perceber que o botão curtir já estará estilizado:



Figura 8.4: <https://goo.gl/OImMvY>

Agora precisamos estilizar as nossas reações. O primeiro passo consiste em esconder a nossa `div` para que esta seja mostrada apenas quando o mouse estiver sob o botão curtir. Conseguimos isso com a propriedade `display` setada para `none`.

Um outro ponto importante é sobre a propriedade `z-index`. Esta é usada para que a `div` `reacoes` fique por cima dos outros conteúdos. Outras estilizações de estilo de borda e alinhamento também devem ser feitas. Veja o trecho de código que as exhibe.

```
.reacoes{
  position: absolute;
  padding: 5px;
```

```

margin-top: -70px;
box-shadow: 1px 1px 2px #cccccc, -1px 0px 2px #eeeeee;
border-radius: 44px 44px;
flex-direction: row;
display: none;
z-index: 1;
}

```

Mudando o valor da propriedade `display`, vamos exibir a div toda vez que o mouse estiver sobre o botão curtir. Veja:

```

.curtir:hover .reacoes {
  display: flex;
}

```

Ao renderizar a página o resultado deve ser similar a:

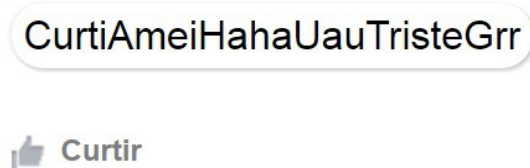


Figura 8.5: <https://goo.gl/L2e6kq>

Chegou o momento de estilizarmos as reações. Cada reação é representada por uma imagem, e usaremos um mapa para defini-las. Acompanhe o trecho de código:

```

$imagens:(
  curti: url(image/curti.png),
  amei: url(image/amei.png),
  haha:url(image/haha.png),
  uau:url(image/uau.png),
  triste:url(image/triste.png),
  grr:url(image/grr.png)
);

```

Como já foi mencionado neste capítulo, um mapa contém uma série de pares de chave e valor. Neste exemplo apresentado, estamos usando o mapa para armazenar o endereço da imagem. Um ponto importante é que você não pode esquecer de colocar as imagens

dentro da pasta `image` . As imagens você pode encontrar no repositório de códigos do livro.

Já temos as imagens, precisamos de espaço para elas. O trecho de código adiante abre espaço para cada reação e garante o seu alinhamento correto.

```
.reacoes{
  ...
  div{
    cursor: pointer;
    width: 40px;
    height: 40px;
    margin: 0 5px;
    display: flex;
    justify-content: center;
  }
}
```

Cada reação possui uma imagem distinta. Vamos usar a função `map-get` que retorna o valor que está associado a chave no mapa. Essa função necessita de dois valores para o seu correto funcionamento: o nome do mapa e a chave onde está o valor a ser recuperado. Acompanhe a seguir:

```
.reacao{
  &Curti{
    background: map-get($imagens, curti);
  }
  &Amei{
    background: map-get($imagens, amei);
  }
  &Haha{
    background: map-get($imagens, haha);
  }
  &Uau{
    background: map-get($imagens, uau);
  }
  &Triste{
    background: map-get($imagens, triste);
  }
  &Grr{
    background: map-get($imagens, grr);
  }
}
```

```
}
```

E ao renderizar a página, temos este resultado:

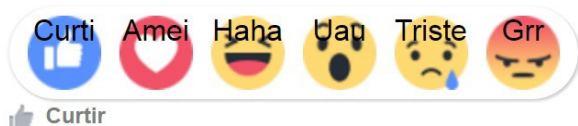


Figura 8.6: <https://goo.gl/1uiMQF>

O resultado já está bem satisfatório. No entanto, precisamos de mais alguns ajustes. As legendas correspondentes a cada reação devem ser estilizadas. No trecho de código a seguir você pode acompanhar uma adição de cor de fundo, troca da cor da fonte, adição de uma borda arredondada, alinhamento e a propriedade `opacity` com o valor 0 para esconder a legenda. Ela só deve ser exibida quando o mouse estiver posicionado na reação.

```
.reacoes{
  ...
  div{
    ...
    span{
      display: block;
      text-align: center;
      position: absolute;
      padding: 3px;
      background-color: #333333;
      border-radius: 10px;
      font-size: 0.8em;
      color: #FFF;
      opacity: 0;
      top: -25px;
    }
    &:hover{
      transform: scale(1.3, 1.3) translateY(-5px);
      span{
        opacity: 0.9;
      }
    }
  }
}
```


No código exibido anteriormente, também temos estilos para o comportamento de hover de cada reação. A propriedade `transform` é responsável por aumentar um pouco a imagem e posicioná-la um pouco acima. Além disso, é possível observar que a opacidade da legenda é modificada para que ela seja exibida no momento em que o mouse estiver posicionado na reação. Veja como deve ficar a renderização na página:

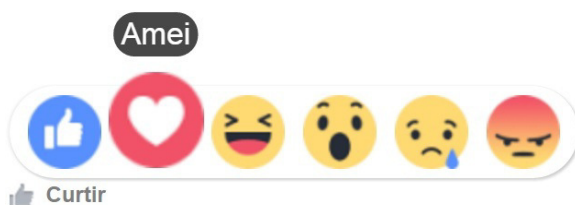


Figura 8.7: <https://goo.gl/L5Zc3L>

O box reações deu um pouco de vida ao nosso projeto. Você pode agora tentar estilizar as ações de compartilhar e comentar.

Você também pode incluir o `mixin` `prefixos` no seu projeto e usá-lo em todas as propriedades que não são suportadas por todos os browsers para que o seu projeto funcione perfeitamente neles. Uma outra ideia é usar as operações matemáticas para calcular o tamanho das suas divs. Mãos à obra!

8.9 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Neste capítulo, foram apresentados os mais variados tipos de dados que são suportados pelo Sass. Já estávamos usando a maioria deles desde o início do livro. No entanto, não tínhamos nos aprofundado em seus conceitos.

Também foi apresentada uma série de funções disponibilizadas para cada tipo de dado e vimos como podemos construir novas funções. Além disso, uma série de operações com os dados foram

mostradas. Essas operações são usadas, em sua maioria, em conjunto com as estruturas de controle, assunto abordado no próximo capítulo.

No capítulo seguinte, você será apresentado ao recurso mais poderoso do Sass: lógica de programação. Usando as estruturas de decisão e repetição apresentadas, seu código ficará extremamente inteligente e reaproveitável. Até lá!

OBTENDO CONTROLE EXTREMO SOBRE O CÓDIGO GERADO

Controlar a saída do código gerado de acordo com o uso de determinados valores é uma técnica extremamente poderosa e deixa o uso dos pré-processadores muito mais interessante. As estruturas de controle constituem um recurso avançado do Sass e requerem um pouco mais de atenção para o seu entendimento.

Se você conhece lógica de programação, vai se sentir mais à vontade com estas estruturas. Caso contrário, está convidado a se aventurar em códigos bem diferentes dos já vistos.

9.1 ESCRREVENDO ESTRUTURAS DE DECISÃO COM @IF E @ELSE E @ELSE IF

Decisões fazem parte de nosso cotidiano. Mesmo fora do mundo dos computadores, estamos sempre tomando decisões. Note: se chover, vou levar o guarda-chuva; ou se sobrar dinheiro, vou comprar um sorvete.

Com as folhas de estilo, não é muito diferente: se usar uma cor de fundo clara, preciso usar uma fonte de cor escura; ou se o botão for pequeno, o tamanho da fonte deve ser pequeno. No entanto, com o CSS nativo, não é possível escrever este tipo de regra. Mas

com o Sass, você pode introduzir condições que serão avaliadas no momento da compilação.

A diretiva `@if` permite adicionar um desvio no código baseado em alguma condição. Isso significa que, ao usá-la, o código compilado vai depender da condição a que a diretiva estava atrelada.

Imagine que, na sua aplicação, os botões possuem como cor de fonte padrão a cor branca. No entanto, no momento da renderização, você percebeu que a cor branca estava prejudicando a usabilidade para alguns botões, já que a leitura ficava bem difícil. Veja na figura a seguir como os botões apareciam no browser:

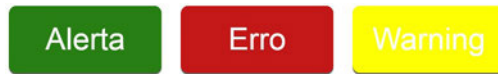


Figura 9.1: <https://goo.gl/1kxIPH>

Pela imagem, é notório que o botão de warning, por possuir a cor amarela como cor de fundo, está com a legibilidade do texto comprometida. Uma das formas de resolver esse problema é trocar a cor da fonte do botão de warning. Mas imagine que você possua inúmeros tipos de botões e, em alguns deles, a fonte de cor branca fica melhor e, em outros, o uso da cor preta para a fonte é mais adequado.

O uso da diretiva `@if`, permite que você adicione uma certa inteligência nas suas regras de estilo. Com ela, é possível que as regras de estilo sejam compiladas com base no resultado de uma expressão.

Voltando ao caso dos botões, você pode verificar se a cor de fundo possui uma alta luminosidade, o que indica que o contraste com a cor branca é menor. E nos casos positivos, trocar a cor de fonte para a cor preta. Acompanhe o código seguinte:

```

@mixin coresBotao ($corFundo){
    $corFonte: #FFFFFF;

    @if(lightness($corFundo) > 50%){
        $corFonte: #000000;
    }

    background-color: $corFundo;
    color: $corFonte;
}

button{
    ...
    &.alerta{
        @include coresBotao(#288014);
    }
    &.erro{
        @include coresBotao(#C41717);
    }
    &.warning{
        @include coresBotao(#FFF03);
    }
}

```

No mixin `coresBotao`, presente no código mostrado anteriormente, temos uma diretiva `@if` que verifica a quantidade de luminosidade que possui a cor de fundo do botão. Vale ressaltar que a expressão usada em conjunto com a diretiva `@if` deve retornar um valor booleano, ou seja, verdadeiro ou falso.

Isso significa que, se a luminosidade da cor de fundo for maior que 50%, a cor da fonte será trocada para preto. Caso contrário, o bloco de comando presente dentro da diretiva `@if` é ignorado e a cor da fonte permanece inalterada.

Uma breve análise do código compilado, exibido a seguir, confirma como funciona a compilação da diretiva `@if`.

```

button.alerta {
    background-color: #288014;
    color: #FFFFFF; }
button.erro {
    background-color: #C41717;
    color: #FFFFFF; }

```

```
button.warning {  
  background-color: #FFFF03;  
  color: #000000; }
```

A figura a seguir exhibe a renderização dos botões com o uso da diretiva `@if` para determinar a cor de fonte deles. Vale ressaltar que essa técnica fica muito mais interessante quando temos um conjunto maior de botões, ou um outro componente qualquer para efetuar o tratamento.

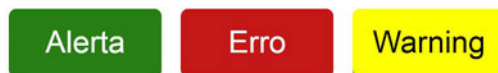


Figura 9.2: <https://goo.gl/eXr8KT>

Vale lembrar que, no Sass, uma estrutura de condição sempre começa com a palavra-chave `@if`, seguida por uma expressão. Essa expressão pode ser uma variável, uma chamada de função ou uma equação, desde que o seu valor seja um booleano. O código executado no caso de a expressão ter valor verdadeiro deve estar entre chaves `{}`.

A diretiva `@if` funciona muito bem em conjunto com as funções de cores, operadores lógicos, relacionais e expressões aritméticas. Em outras palavras, você pode fazer uso dessa diretiva para calcular tamanho e peso de fontes, espaçamentos, estilos para as bordas, sombras, enfim, o que sua imaginação permitir.

Em conjunto com a diretiva `@if`, opcionalmente, você pode incluir a diretiva `@else` imediatamente após. O bloco de estilo que compõe a diretiva `@else` é compilado apenas se a expressão da diretiva `@if` for falsa.

Isso significa que, quando as diretivas `@if` e `@else` são usadas em conjunto, podemos expressar duas alternativas para a compilação. Uma delas executa caso a condição seja verdadeira, e a outra quando a condição não for satisfeita. A imagem adiante ilustra

como funciona o processo em uma estrutura condicional.

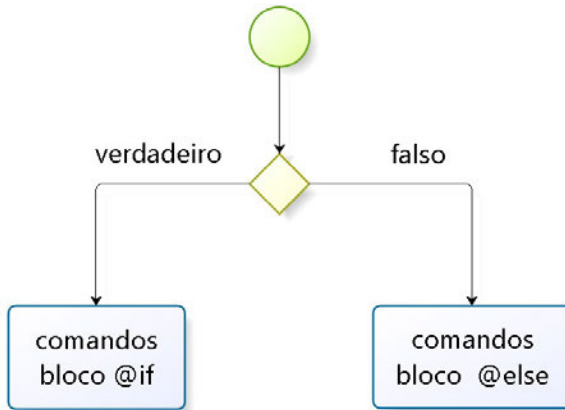


Figura 9.3: <https://goo.gl/wWaU95>

O Sass também possui a diretiva `@else if` que, como o nome sugere, é uma combinação das duas instruções. Essa diretiva deve ser usada em conjunto com a diretiva `@if` e será executada apenas se a expressão do `@if` não for satisfeita.

Entretanto, diferentemente do `@else`, o `@else if` possui uma expressão. Em outras palavras, se a expressão do `@if` for falsa, a expressão do `@else if` será experimentada. Caso ela seja verdadeira, o bloco de estilo aninhado à diretiva é compilado. Caso a expressão seja falsa, o compilador do Sass procura pelo próximo `@else if`. Acompanhe a seguir a sintaxe de um bloco de decisão no Sass.

```
@if(expressao booleana){
  bloco de estilo
}@else if(expressao booleana){
  bloco de estilo
}@else if(expressao booleana){
  bloco de estilo
...
}@else if(expressao booleana){
  bloco de estilo
}@else{
```

```
    bloco de estilo  
}
```

Vale ressaltar que é possível usar diversos `@else if` para o mesmo `@if` e apenas um `@else`. Ou seja, o seu bloco de decisão deve iniciar sempre com a diretiva `@if`. Após a diretiva `@if`, você pode usar a diretiva `@else if` ou `@else`. Essas duas últimas são opcionais. Isso significa que você pode ter um bloco de decisão com apenas `@if`, com `@if` e `@else if`, com `@if` e `@else`, e ainda a mistura das três: `@if`, `@else if` e `@else`.

É importante enfatizar que o uso da diretiva `@else if` aumenta o número de alternativas para a compilação. Em outras palavras, a expressão de uma diretiva `@else if` é avaliada apenas se as diretivas `@if` ou `@else if` anteriores a ela forem falsas.

O `@else` deve ser sempre o último bloco. Ele terá seu bloco de estilo compilado se todas as outras diretivas da estrutura de decisão tiverem suas expressões com valor falso.

Normalmente, as estruturas de decisão são usadas em conjunto com mixins e functions, visando aumentar o poder de reutilização deles, visto que o comportamento pode ser alterado de acordo com o valor dos parâmetros recebidos nas functions e mixins.

Para ilustrar a utilização de uma estrutura de decisão um pouco mais complexa, vamos construir um mixin para desenhar uma setinha na página. Uma seta normalmente serve para indicar alguma informação adicional, ou um caminho a seguir, e tem uma direção. A ideia é construir um mixin que desenhe uma seta usando direção, tamanho e cor variáveis.

Acompanhe a seguir a construção do mixin. Veja que inicialmente foram setados os valores que são iguais para as setas de qualquer direção, como o tamanho da borda, a cor da borda, entre outros. Em seguida, foi adicionado um bloco de decisão.

A diretiva `@if` possui os comandos para a criação de uma seta que aponte para a direita. Isso indica que, se no `include` do `mixin` o valor da variável `$direcao` for direita, o bloco de código aninhado a essa diretiva será adicionado ao CSS, e os demais blocos `@else if` e `@else` devem ser ignorados.

O mesmo ocorre para os `@else if` para a direção esquerda e topo. No final, um bloco `@else` deve desenhar uma seta para baixo nos casos em que qualquer coisa diferente de esquerda, direita ou topo for adicionada como direção.

```
@mixin setinha($tamanho, $direcao, $cor){
  border-width: $tamanho;
  border-color: transparent;
  border-style: solid;
  display: inline-block;
  height: 0;
  width: 0;

  @if($direcao == "direita"){
    border-left-color: $cor;
    border-right-width: 0;
  }@else if($direcao == "esquerda"){
    border-right-color: $cor;
    border-left-width: 0;
  }@else if($direcao == "topo"){
    border-bottom-color: $cor;
    border-top-width: 0;
  }@else{
    border-top-color: $cor;
    border-bottom-width: 0;
  }
}

div{
  @include setinha(20px, direita, red);
}
```

Note no CSS mostrado a seguir como ficou a compilação do `mixin`. No exemplo dado, a construção da seta deveria ser à direita, então apenas os comandos que estavam na diretiva com a expressão `@direcao == "direita"` foram compilados.

```
div {
  border-width: 20px;
  border-color: transparent;
  border-style: solid;
  display: inline-block;
  height: 0;
  width: 0;
  border-left-color: red;
  border-right-width: 0; }
```

A renderização no browser deve ficar similar à figura:



Figura 9.4: <https://goo.gl/khBmZp>

Com os exemplos apresentados, é possível ter uma dimensão do poder de utilização de uma estrutura de decisão. Seu uso permite a criação de mixins muito mais versáteis, aumentando o poder de reutilização ao extremo.

9.2 MONTANDO UMA ESTRUTURA DE CONTROLE COM @FOR

A diretiva `@for` é usada quando é necessário repetir um bloco de estilo em uma quantidade limitada de vezes. Por este motivo, também é chamado de laço de repetição, e está presente na grande maioria das linguagens de programação.

Um laço normalmente é usado para iterar sobre uma coleção de elementos e realizar ações similares em todos os membros da coleção. Observe a figura a seguir:



Figura 9.5: <https://goo.gl/0CL3ZB>

Essa figura mostra um menu, em que cada item tem uma pequena variação no tom da cor de fundo. Para construí-lo, a primeira opção que vem a mente é setar a cor de fundo de cada item do menu, como mostrado no CSS a seguir.

```
.menu a:nth-child(1) {  
    background: #0e07e1;  
}  
.menu a:nth-child(2) {  
    background: #5e07e1;  
}  
.menu a:nth-child(3) {  
    background: #ae07e1;  
}  
.menu a:nth-child(4) {  
    background: #e107c4;  
}  
.menu a:nth-child(5) {  
    background: #e10774;  
}
```

Note que o código exibido anteriormente é bastante repetitivo. Se considerarmos um menu com mais itens, a quantidade de linhas de código aumentará proporcionalmente. E se o nosso cliente pedir para trocar a cor base do degradê, teremos um grande trabalho a ser feito se construirmos o estilo dessa forma, digamos, primordial.

Considerando que códigos inteligentes são mais fáceis de manter, é possível construir este menu, usando a diretiva `@for` em conjunto com a função de cor `adjust-hue`, apresentada no capítulo anterior.

Para conseguir este efeito, é necessário entender como a diretiva funciona. Primeiramente, existem duas formas distintas de trabalhar com esta diretiva. Veja:

```
@for $i from <inicio> through <fim>{  
    ...  
}  
  
@for $i from <inicio> to <fim> {  
    ...
```

```
}
```

Note que, em ambas sintaxes, a diretiva `@for` necessita de três valores numéricos para a sua execução: um contador, o valor inicial e o valor final do loop.

O contador representado pela variável `$i` indica o estado corrente do loop. Ou seja, a cada iteração, o contador é incrementado. Para cada repetição, a variável `$i` é usada para ajustar o código de saída. O nome `$i` é uma convenção adotada pelos programadores. No entanto, você pode usar qualquer outro nome, se preferir.

A condição de início define o valor inicial do loop. Ou seja, o valor na primeira iteração e a condição de fim indicam quando a iteração deve parar.

Comparando as duas sintaxes, é possível perceber que a diferença entre elas está no uso da palavra `through` ou `to`. A primeira opção (`through`) executa o loop até que o valor do contador seja igual a condição de fim. Isso significa que, se tivermos os valores 1 e 4 como valores de início e fim, respectivamente, quatro seletores CSS devem ser montados. Acompanhe o exemplo a seguir para ficar mais claro.

```
@for $i from 1 through 4 {  
  .teste-#{ $i } {  
    width: 50px + $i;  
  }  
}
```

Nesse código, temos uma diretiva `@for` que itera dos valores 1 até 0 4. Para cada iteração, um seletor CSS é criado usando uma interpolação entre a palavra `teste-` e o valor do contador. No seletor criado, é adicionada a propriedade `width` com o valor de 50px adicionado ao valor do contador. Veja adiante como deve ficar o CSS gerado.

```
.teste-1 {
  width: 51px; }

.teste-2 {
  width: 52px; }

.teste-3 {
  width: 53px; }

.teste-4 {
  width: 54px; }
```

Note que são criados quatro seletores usando a interpolação. Também é possível verificar que o incremento é no valor de uma unidade.

Já a segunda opção — usando a palavra `to` — executa o loop enquanto o contador for menor que a condição de fim. Isso significa que, se usarmos os valores 1 e 4 como condição de início e fim, o loop será executado três vezes, produzindo três seletores CSS. Veja o trecho de código a seguir.

```
@for $i from 1 to 4 {
  .teste-#{ $i } {
    width: 50px + $i;
  }
}
```

No CSS exibido a seguir, é possível notar a diferença. Perceba que apenas três seletores foram criados.

```
.teste-1 {
  width: 51px; }

.teste-2 {
  width: 52px; }

.teste-3 {
  width: 53px; }
```

No Sass, o valor inicial pode ser maior do que o final. Neste caso, em vez de incrementar uma unidade, o contador decrementa. Veja o exemplo:

```
@for $i from 5 to 2 {
  .teste-#{ $i } {
    width: 50px + $i;
  }
}
```

O CSS gerado é similar ao exibido a seguir:

```
.teste-5 {
  width: 55px; }

.teste-4 {
  width: 54px; }

.teste-3 {
  width: 53px; }
```

Neste momento, você já deve ter uma ideia de como montar o menu de uma forma mais sofisticada. Bom, existem diversas formas de construir um menu e, neste exemplo, o menu será criado a partir de um conjunto de links. Veja a estrutura do HTML.

```
<div class="menu">
  <a href="#">Lançamento</a>
  <a href="#">Marca</a>
  <a href="#">Coleção</a>
  <a href="#">Imprensa</a>
  <a href="#">Contato</a>
</div>
```

Observando atentamente o HTML, podemos notar a presença de um conjunto de links ou uma coleção de links. Para iterar sobre ela, podemos usar a pseudoclasse `:nth-child`. Aproveitando o ensejo, vale ressaltar que a diretiva `@for` é especialmente útil para ser usada com as pseudoclasses `:nth-child`, `:nth-of-type`, `:nth-last-child` e `:nth-last-of-type`.

Observe o código seguinte, usado para montar o menu exibido no início do tópico:

```
$corBase: #0750E1;
.menu{
  line-height: 30px;
```

```

display: inline-flex;
border-bottom: 1px solid #ACABAE;
a{
    padding: 10px 20px;
    text-decoration: none;
    color: #FFFFFF;
    border-left: 1px solid #FFFFFF;
    font-size: 18px;
    border-radius: 10px 10px 0 0;
    @for $i from 1 through 5 {
        &:nth-child(#{ $i }) {
            $corItem: adjust-hue($corBase, ($i * 22dg));
            background: $corItem;
            &:hover{
                background: darken($corItem, 20%);
            }
        }
    }
}

```

Vamos concentrar nossa análise no uso da diretiva `@for`. Observe que, na declaração, temos como valor inicial 1, que representa o primeiro item do menu, e o valor final 5, que representa o último item de menu. Isso significa que o valor final será sempre igual a quantidade de itens do menu.

Para estilizar cada item com uma cor diferente, foi usada a pseudoclasse `:nth-child`. Note que o índice usado por ela é incrementado através da interpolação com a variável `$i`. Portanto, na primeira iteração, teremos `:nth-child(1)`, na segunda `:nth-child(2)`, e assim sucessivamente até chegar ao índice 5.

Vale observar também que, para montar a variação no tom da cor, foi usada a função `adjust-hue` com uma cor de base e uma variação no tom, calculada pela variável `$i` multiplicada por 22dg. Como variável `$i` é incrementada em uma unidade a cada iteração, cada item do menu terá uma diferença de 22dg na roda de cores apresentada no capítulo anterior.

O estilo do comportamento de `:hover` também foi montado

de uma forma inteligente. Note que foi usada a função `darken` , que escurece a cor, usando um escurecimento de 10% a partir da cor do item de menu.

A parte mais interessante de tudo isso é que podemos construir menus com cores totalmente diferentes mudando apenas a cor de base e a diferença do ângulo. Também é possível usar a mesma ideia com outras funções de cores, como o `saturate` ou `transparentize` .

Para melhorar o entendimento, vale uma olhadinha no código compilado:

```
.menu a:nth-child(1) {  
  background: #0e07e1; }  
.menu a:nth-child(1):hover {  
  background: #08047e; }  
.menu a:nth-child(2) {  
  background: #5e07e1; }  
.menu a:nth-child(2):hover {  
  background: #35047e; }  
.menu a:nth-child(3) {  
  background: #ae07e1; }  
.menu a:nth-child(3):hover {  
  background: #61047e; }  
.menu a:nth-child(4) {  
  background: #e107c4; }  
.menu a:nth-child(4):hover {  
  background: #7e046e; }  
.menu a:nth-child(5) {  
  background: #e10774; }  
.menu a:nth-child(5):hover {  
  background: #7e0441; }
```

A diretiva `@for` também pode ser usada para a criação de grids. Os sistemas de grid, atualmente, são uma excelente opção para a criação de layouts estruturais. Com eles, é possível aumentar a produtividade e são ideais para telas responsivas. Neste livro, será feito um sistema de grids com a diretiva `@while` , que é o próximo tópico. Baseado neste exemplo, você pode construí-lo com a diretiva `@for` .

9.3 CONTROLANDO A SAÍDA DE CÓDIGO COM @WHILE

A diretiva `@while` também é um laço de repetição, no qual um determinado bloco de código é repetido enquanto uma condição for verdadeira. Ele é muito similar ao `@for`, no entanto, as iterações devem ser controladas manualmente.

Acompanhe como a diretiva `@while` pode ser usada para a criação do mesmo menu exibido no tópico referente a diretiva `@for`.

```
$i: 1;
@while ($i < 6) {
    &:nth-child(#{ $i }) {
        $corItem: adjust-hue($corBase, ($i * 22dg));
        background: $corItem;
        &:hover{
            background: darken($corItem, 20%);
        }
    }
    $i: $i + 1;
}
```

Note no trecho de código exibido que há uma variável `$i`, declarada com o intuito de controlar o bloco de repetição. Ela possui o valor inicial de 1. Na condicional da diretiva `@while`, é verificado se o `$i` possui valor menor do que 6, condição primordial para o bloco ser executado.

Note que o incremento da variável `$i` é feito manualmente em `$i: $i + 1`. A diretiva `@while` não realiza o incremento automaticamente.

As estruturas de repetição como o `@for` e o `@while` são excelentes para a criação de sistemas de grid. Acompanhe a criação de um sistema de grid com 12 colunas.

```
.linha{
```

```

    display: flex;
}

$i: 1;
@while($i <= 12){
    .painel-#{ $i }{
        flex: $i;
    }
    $i: $i + 1;
}

```

Note que a grid foi construída apenas setando a propriedade `flex` com o tamanho correspondente de cada painel. O código compilado fica similar ao mostrado a seguir:

```

.painel-1 {
    flex: 1; }

.painel-2 {
    flex: 2; }

.painel-3 {
    flex: 3; }

.painel-4 {
    flex: 4; }

...

.painel-12 {
    flex: 12; }

```

Para fazer uso desse sistema de grid, é necessário seguir uma estrutura de HTML que contemple as classes `painel` e `linha`. Veja:

```

<div class="linha">
    <div class="painel-9">Painel 9</div>
    <div class="painel-3">Painel 3</div>
</div>
<div class="linha">
    <div class="painel-12">Painel 12</div>
</div>
<div class="linha">
    <div class="painel-4">Painel 4</div>
    <div class="painel-4">Painel 4</div>
    <div class="painel-2">Painel 2</div>
    <div class="painel-1">Painel 1</div>

```

```
<div class="painel-1">Painel 1</div>
</div>
```

E o resultado no navegador deve ficar similar à figura:



Figura 9.6: <https://goo.gl/2oqBK5>

Note que uma infinidade de layouts pode ser construída usando esse sistema de grid. Você também pode variar a quantidade de colunas e o tamanho da margem usada.

A diretiva `@while` é uma opção para a diretiva `@for`. Tudo o que você consegue fazer com uma, conseguirá com a outra também. A escolha por uma delas trata-se apenas de uma opção pessoal.

9.4 MONTANDO CLASSES DINAMICAMENTE COM `@EACH`

Imagine que você tenha várias imagens que precisa incluir na sua folha de estilo. e todas elas estão na mesma pasta, a única diferença entre elas é o nome. Normalmente, isso é feito para cada seletor separadamente. No entanto, com o Sass, é possível escrever usando a estrutura de repetição `each`.

A diretiva `@each` também é uma estrutura de repetição que possui como objetivo iterar sobre os elementos de uma coleção, que pode ser definida em forma de lista ou como mapa. Sendo assim, para cada item da coleção, são produzidos blocos de código usando os valores do item. Isso significa que o loop é executado na mesma

quantidade de vezes que o tamanho da coleção.

Para exemplificar o uso do `@each`, vamos construir uma barra de redes sociais. Inicialmente, colocaremos na pasta `images` um ícone para cada rede social presente na barra. No nosso exemplo, temos o Facebook, Twitter, LinkedIn, Instagram e Pinterest. Lembre-se de que os nomes das imagens devem ser exatamente o mesmo da rede social correspondente.

Em seguida, vamos criar uma lista que contenha todas as redes sociais. Você pode adicionar outras de sua preferência. No nosso exemplo, nomeamos a lista como `$redes`.

O código seguinte exibe como a lista foi iterada. Veja que, após o comando `@each`, usamos uma variável chamada `$rede`. Essa variável representa o item da lista que está na iteração do momento. Usamos também a variável `$redes` que, no exemplo apresentado, armazena os nomes das redes sociais a serem iteradas.

Outro ponto a observar é que estamos fazendo uso da interpolação, vista no capítulo anterior, para criar o seletor e também para preencher o caminho da imagem.

```
$redes: facebook, twitter, linkedin, instagram, pinterest;
@each $rede in $redes {
  .#{$rede}{
    background: url("image/#{$rede}.png");
  }
}
```

Acompanhando o código compilado, fica mais claro como o comando `@each` trabalha:

```
.facebook {
  background: url("image/facebook.png"); }

.twitter {
  background: url("image/twitter.png"); }

.linkedin {
```

```
background: url("image/linkedin.png"); }

.instagram {
  background: url("image/instagram.png"); }

.pinterest {
  background: url("image/pinterest.png"); }
```

Renderizando no browser, você deve ter uma barra de redes sociais similar à apresentada a seguir.



Figura 9.7: <https://goo.gl/hw0OAH>

9.5 UM PROJETO REAL

As estruturas de repetição apresentadas neste capítulo são extremamente úteis para a criação de grids, como já foi mencionado. Sendo assim, vamos construir uma estrutura de grid e aplicá-la no cabeçalho do nosso projeto.

A montagem de uma estrutura de grid é relativamente simples ao usar uma estrutura de repetição aliada ao `flexbox`. Veja:

```
.linha{
  display: flex;
}

[class*="painel"]{
  margin: 5px;
  display: flex;
}

$i: 1;
@while($i <= 12){
  .painel-#{ $i }{
    flex: $i;
  }
  $i: $i + 1;
}
```

Acompanhando o código apresentado anteriormente, podemos destacar alguns pontos importantes. O primeiro deles é a estilização da linha que representa um container onde ficarão dispostos os painéis que devem ser numerados de 1 até 12, usando o recurso de interpolação. O segundo ponto é que a numeração representa a quantidade de colunas que o painel deve ocupar. Usamos a propriedade flex para definir o seu tamanho.

Vale lembrar de que essa estrutura de grid pode ser usada em diversas partes da aplicação. Dessa forma, é interessante que ela fique em um arquivo separado.

O esqueleto do HTML para a montagem do cabeçalho, usando a estrutura de grids, deve ser similar ao apresentado adiante.

```
<div class="cabecalho">
  <div class="linha">
    <div class="painel-6"></div>
    <div class="painel-2"></div>
    <div class="painel-2"></div>
    <div class="painel-2"></div>
  </div>
</div>
```

Veja no HTML que temos uma div chamada `linha` que possui quatro painéis. Vale lembrar de que, para que a estrutura de grid funcione perfeitamente, é necessário que a soma dos painéis seja 12. No nosso caso, por exemplo, temos um painel com tamanho 6 e três painéis com tamanho 2 ($6 + 2 + 2 + 2 = 12$).

Ao renderizar em um navegador, o exemplo apresentado deve ficar semelhante à figura a seguir. É importante ressaltar que o uso de grids para montagem de um container permite a criação de layouts fluidos. Ou seja, se você diminuir ou aumentar a resolução, os painéis vão se adaptar e preencher o espaço disponível.



Figura 9.8: <https://goo.gl/96XHwS>

Pela imagem, é possível perceber que já temos uma base para colocarmos os itens de cabeçalho. Vamos agora incrementar o HTML com alguns itens. Acompanhe:

```
<div class="cabecalho">
  <div class="linha">
    <div class="painel-6">
      <div class="icone"></div>
      <div class="busca"></div>
    </div>
    <div class="painel-2">
      <div class="usuario">Rosi</div>
      <div class="divisorVertical"></div>
      <div class="pgInicial">Página inicial</div>
    </div>
    <div class="painel-2">
      <div class="pessoa"></div>
      <div class="mensagem"></div>
      <div class="mundo"></div>
    </div>
    <div class="painel-2"></div>
  </div>
</div>
```

Para que os itens fiquem dispostos de forma centralizada, precisamos adicionar mais algumas estilizações. Acompanhe:

```
.linha{
  height: 24px;
  align-items: center;
  margin-top: 5px;
}

[class*="painel"]{
  align-items: center;
}
```

Agora precisamos adicionar os estilos de cada item do cabeçalho. Para facilitar essa tarefa, construiremos um mixin com os padrões que são comuns à maioria desses itens.

```
@mixin itensCabecalho{
  height: 24px;
  width: 24px;
  border-radius: 2px;
```

```
}
```

Precisamos também adicionar as imagens que são usadas para a criação dos ícones. Veja que, além de adicionar a imagem, usamos o mixin criado para determinar o tamanho:

```
.pessoa{
  background: url(image/pessoas.png);
  margin-right:10px;
  @include itensCabecalho;
}

.mensagem{
  background: url(image/mensagem.png);
  @include itensCabecalho;
  margin-right:10px;
}

.mundo{
  background: url(image/mundo-cabecalho.png);
  @include itensCabecalho;
}
```

Por fim, a estilização dos demais itens. Em alguns deles, também fizemos uso do mixin criado anteriormente. As demais estilizações estão relacionadas a fontes, tamanhos e cores. Nada muito diferente do que já vimos em todos os outros capítulos do livro.

```
.icone{
  background: $corFundoBoxes;
  @include itensCabecalho;
}

.busca{
  margin-left: 10px;
  @include itensCabecalho;
  width: 450px;
  background: $corFundoBoxes;
}

.usuario{
  color: $corFonteDestaque;
  &:before{
    content: "";
    display: inline-block;
    vertical-align: middle;
```



```

        margin-right: 5px;
        @include itensCabecalho;
        background: #000000;
    }
}

.pgInicial{
    color: $corFonteDestaque;
}

.divisorVertical{
    height: 20px;
    border-left: 1px solid #365089;
}

```

A renderização da página neste momento deve ficar similar à apresentada a seguir. Você pode perceber que os itens ficaram dispostos em cada painel. E o mais legal é que funciona para diversos tamanhos de tela.

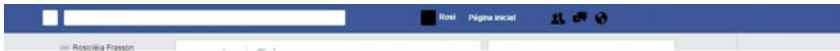


Figura 9.9: <https://goo.gl/7K64UQ>

Um outro ponto a ressaltar é que as grids podem ser construídas com a quantidade de colunas que você desejar. Não necessariamente deve ser 12. Podemos construí-las com a quantidade que mais se adapte ao nosso layout.

9.6 O QUE ESPERAR DO PRÓXIMO CAPÍTULO

Neste capítulo, tivemos a oportunidade de explorar ao máximo os recursos do Sass. Usar lógica de programação para escrever CSS é uma ótima solução para automatizar tarefas repetitivas na escrita de folhas de estilo de uma forma inteligente.

Com este recurso, você também está apto a construir mixins e funções muito mais adaptáveis. Dessa forma, você pode construir a sua própria biblioteca e usá-la nos mais diferentes projetos. Você

pode também compartilhá-la com a comunidade no GitHub.

Infelizmente, nossa jornada está chegando ao fim. No próximo e último capítulo, você vai conhecer um pouquinho do universo em torno do Sass com a apresentação de alguns plugins. Também no capítulo seguinte, vou deixar uma lista de referências que podem ser uma ótima fonte de consulta para os seus projetos com Sass.

INDO ALÉM

No transcorrer deste livro, você foi apresentado a uma série de funcionalidades que podem ajudá-lo a escrever CSS de uma forma muito mais inteligente, deixando as tarefas repetitivas e entediantes para o pré-processador. No entanto, além das funcionalidades inclusas na biblioteca padrão, existe um ecossistema em torno do Sass, que pode ser muito útil na construção de folhas de estilo.

Embora o foco deste livro tenha sido explorar as funcionalidades já inclusas no Sass, acredito que será muito vantajoso você ter, ao menos, conhecimento da existência de algumas extensões que podem ser adicionadas ao seu projeto para a adição de funcionalidades ao Sass e aumentar ainda mais a sua produtividade.

10.1 ECOSSISTEMA SASS

Durante os anos de existência do Sass, desenvolvedores foram criando bibliotecas de funcionalidades extras ao Sass e transformaram muitas delas em projetos open-source. Uma rápida pesquisa no GitHub por Sass vai retornar mais de 19.000 resultados.

Isso significa que se você precisar de extensões para resolver um problema, terá várias opções para escolher. E mais, se não encontrar nenhuma solução que lhe atenda, pode desenvolver e compartilhar com a comunidade.

Os próximos tópicos descrevem algumas extensões mais

conhecidas do Sass com um breve relato do que elas podem lhe oferecer.

Compass

O Compass foi lançado em 2009 e foi projetado com o intuito de gerenciar pacotes Sass, incentivando o compartilhamento de projetos open-source. No entanto, o Compass tornou-se popular por sua vasta biblioteca de mixins vendor prefix.

O Compass fornece ainda funções extras de cores e uma série de mixins que resolvem problemas conhecidos pelos desenvolvedores, como reset, clear-fix, sprite, truncate e montagem de gradientes.

Para mais informações sobre o Compass, você pode visitar a página do projeto: <http://compass-style.org/>

Bourbon

O Bourbon é uma biblioteca mais leve que pode ser usada como alternativa ao Compass. Ela também possui suporte aos vendor prefix e uma quantidade enorme de mixins para reset, clear-fix, truncate, box-sizing, keyframes, placeholder, entre outros. O Bourbon também oferece um sistema de grid próprio, o neat.

Você encontra a documentação do Bourbon em: <http://bourbon.io/>.

Susy, Zen Grids e Blueprint

Existe uma infinidade de bibliotecas para auxiliar na construção de grids responsivas. Dentre as mais conhecidas, estão a Susy (<http://susy.oddbird.net>), Zen Grids (<http://zengrids.com/>) e Blueprint (<http://compass-blueprint.org>).

Basicamente, para fazer uso de uma biblioteca de grids, você

precisa seguir a estrutura de HTML imposta pela biblioteca com a adição de algumas classes CSS.

Essas bibliotecas geralmente possuem variações na quantidade de colunas usadas para a construção do layout e espaçamentos configuráveis. Podem ser uma ótima solução para a construção de layouts responsivos de forma ágil.

10.2 NÃO PARE POR AQUI

Neste livro, foram abordadas as principais funcionalidades do Sass. Tive a preocupação em mostrar a utilidade de cada uma em projetos reais para facilitar seu aprendizado no transcorrer da leitura.

No entanto, o universo em torno dele é muito amplo, sendo impossível abordar todas as particularidades em um único livro. Além disso, as mudanças no mundo tecnológico acontecem de forma muito rápida e você precisa acompanhá-las.

Sendo assim, a seguir estão listadas algumas referências que me foram úteis e acredito que podem lhe servir também. Infelizmente os links estão em inglês, mas são excelentes materiais.

- Página da documentação oficial. Nela você encontra todas as funcionalidades do Sass — <http://sass-lang.com/documentation/>
- Diversos tutoriais e artigos desde níveis iniciais até avançados — <http://thesassway.com/>
- Uma página com podcasts, entrevistas e tutoriais sobre Sass — <https://www.youtube.com/user/sassbites/videos>
- Uma série de convenções para escrita de código Sass — <https://sass-guidelin.es/>

- Página do Sass no Twitter. Nela você tem acesso às últimas novidades — <https://twitter.com/SassCSS>
- Possui muitos artigos sobre CSS no geral. É frequentemente atualizado e possui uma gama enorme de assuntos abordados — <https://www.smashingmagazine.com/>
- Página com uma infinidade de artigos sobre o mundo front-end. Excelente fonte de consulta — <https://css-tricks.com/>

10.3 PALAVRAS FINAIS

Escrever este livro foi muito gratificante e divertido. Espero que você tenha curtido a leitura tanto quanto eu curti a escrita. Também espero que, além de agradável a leitura, tenha sido muito proveitosa.

E, para finalizar, agradeço por ter dedicado seu tempo na leitura deste livro. Obrigada!